НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМ. ІГОРЯ СІКОРСЬКОГО»
Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

.

Дипломна робота

на тему: Розробка програмного агента моніторингу та управління теплонасосної установки

Студент групи ТМ-51 Десятник Олександр Станіславович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник роботи Ст. викладач Мірошниченко І.В. _____
(вчені ступінь та звання, прізвище, ініціали) (підпис)

Кількість сторінок **70**

Кількість ілюстрацій **19**

Київ – 2019

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
_____ О.В.Коваль
(підпис) (ініціали, прізвище)
"___"_____2019 р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки
6.050101 "Комп'ютерні науки''

на тему: Розробка програмного агента моніторингу та управління теплонасосної установки

Виконав: студент 4 курсу, групи ТМ-51

Десятник Олександр Станіславович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник Ст. викладач Мірошниченко І.В. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019
Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки 6.050101 “Комп’ютерні науки’’

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2019 р.

ЗАВДАННЯ

на дипломну роботу студенту

Десятнику Олександру Станіславовичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Розробка програмного агента реєстру мультиагентної системи”

керівник роботи Ст. викладач Мірошниченко І.В.
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”___”_____ 201__ р. № _____

2. Строк подання студентом роботи ___ _____ 201__ р.

3. Вихідні дані до роботи журнал історії комунікації агентів, журнал помилок системи

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) проаналізувати існуючі програмні рішення та засоби розробки програмного агента реєстру мультиагентної системи, розробити програмний агент, який у реальному підтримує стабільність роботи мультиагентної системи, забезпечує агентів можливістю комунікації, веде журнали історії виконаних операцій та історії помилок

5. Перелік ілюстраційного матеріалу (з точним зазначенням обов’язкових креслень) 1.

Актуальність 2. Мета і завдання роботи 3. Огляд існуючих програмних рішень 4. Задачі програмного агента 5. Структура проекту та використані технології 6. Структура бази даних 7. Алгоритм роботи системи. 8.Приклад виводу консолі сканера мережі. 9. Приклад взаємодії агента із сервером MAC. 10. Приклад взаємодії агентів через сервер MAC. 11.Висновки.

Дата видачі завдання ” 1 ” грудня 2018 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Вивчення та аналіз задачі	09.10.18	
2.	Розробка архітектури та загальної структури системи	05.02 – 11.02.19	
3.	Розробка структур окремих підсистем	12.02 – 18.02.19	
4.	Підготовка матеріалів	19.02 – 25.02.19	
5.	Програмна реалізація системи	26.02 – 6.03.19	
6.	Захист програмного продукту	12.04 – 30.05.19	
7.	Оформлення пояснювальної записки	18.05.18	
8.	Передзахист	01.06.19	
9.	Захист		

Студент _____ Десятник О.С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Мірошниченко І.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Пояснювальна записка містить 51 сторінку, включає 19 рисунків, 10 таблиць та 24 посилання.

Метою дипломної роботи є створення програмного агента реєстру мультиагентної системи. Було створено серверний додаток за допомогою мови програмування C# у інтегрованому середовищі розробки Microsoft Visual Studio. У якості системи управління базами даних використано Microsoft SQL Server.

Розроблено програмний агент реєстру мультиагентної системи з функціями забезпечення стабільності функціонування та відмовостійкості МАС, ведення журналів виконаних операцій та історії помилок, забезпечення комунікації агентів МАС, моніторингу мережі на доступність агентів та внесення їх до реєстру.

Ключові слова: мультиагентна система, програмний агент, “Yellow Pages”, C#, Socket, TCP.

ABSTRACT

The explanatory note contains 51 pages, including 19 figures, 10 tables and 24 references.

The purpose of the thesis is to create a software agent for the registry of the multi-agent system. A server application was created using the C # programming language in the integrated Microsoft Visual Studio development environment. Microsoft SQL Server is used as a database management system.

The software agent of the multiagent system registry system has been developed with the functions of ensuring the stability of the operation and fault tolerance of MAS, maintaining logs of operations performed and error history, providing communication of MAC agents, monitoring the network for the availability of agents and adding them into the registry.

Keywords: multi-agent system, software agent, "Yellow Pages", C#, Socket, TCP.

ЗМІСТ

ВСТУП

З розвитком та ускладненням програмних комплексів виникла необхідність спрощення представлення складних систем та урахування в них можливих дій користувача. Одним із ефективних способів вирішення цієї проблеми став мультиагентний підхід до моделювання складних систем. Але на цей час розроблені процедури проектування складних мультиагентних систем додатково ускладнюють власне сам процес проектування, пропонуючи використовувати різнорідні об’єкти, окрім самих агентів.

У першому розділі пояснювальної записки описано призначення системи, яка розробляється та постановку задачі програмного забезпечення.

В другому розділі записки міститься опис предметної області та огляд існуючих програмних рішень для поставленої задачі.

В третьому розділі проводиться обґрунтування методів та прикладних засобів програмування поставленої задачі.

Четвертий розділ є документальним супроводом програмного забезпечення. Він містить інформацію про складові компоненти програмного забезпечення та їх взаємодію.

В п’ятому розділі описується методика роботи користувача з програмною системою.

1. ЗАДАЧА РОЗРОБКИ ПРОГРАМНОГО АГЕНТА РЕЄСТРУ МУЛЬТИАГЕНТНОЇ СИСТЕМИ

Метою цієї роботи є створення системи, яка б вела реєстр інтелектуальних агентів, публікувала дані про склад та доступні сервіси системи, обробляла запити від агентів та зберігала історію комунікації агентів.

Програмний комплекс має вирішувати наступні задачі:

– забезпечення стабільності функціонування та відмовостійкості МАС;
– ведення журналів виконаних операцій та історії помилок;
– забезпечення комунікації агентів МАС;
– моніторинг мережі на доступність агентів та внесення їх до реєстру.

Для обробки повідомлень від агентів необхідно створити TCP сервер[1], який буде перевіряти можливість виконання команд та зберігати історію комунікації агентів. При отриманні повідомлення від агента програма повинна перевіряти його на правильність побудови за заздалегідь визначеним синтаксисом. У разі невірного запиту сервер має повертати повідомлення у якому міститься код помилки. Сервер повинен показувати повідомлення про кожну отриману команду від агентів. Повідомлення повинно містити дані про агента та текст запитуваної команди. Сервер повинен надсилати відповідь на кожен вхідний запит, яка повинна містити результат запиту, або код помилки, у разі невдалого виконання запиту.

Програмний продукт повинен бути написаний за допомогою техніки розробки програмного забезпечення DDD та технології .NET Core[2]. Підтримувати платформу Linux ARM.

Користувачами системи являються агенти мультиагентної системи. Інтерфейс користувача реалізовано за шаблоном консольного додатку, оскільки користувачі напряму із ним не взаємодіють а спілкування між агентами здійснюється за допомогою повідомлень на базі сокетів по каналу TCP у форматі символьних рядків за заздалегідь

визначеним синтаксисом. Вхідною інформацією є рядок символів який транслюється у команду зрозумілу для сервера.

Вихідною інформацією є результат виконаної команди на сервері чи іншому агенті. Якщо команда адресована іншому пристрою, сервер повертає код відповіді та, якщо передбачено, результат виконання команди. Якщо сервер отримав команду яка не підтримується він повертає повідомлення з відповідним текстом. У разі виникнення непередбачуваної помилки сервер повертає повідомлення з її кодом та описом, а також доповнює журнал помилок.

Потенційними сферами застосування даної системи можуть бути мультиагентні системи які в якості каналу зв’язку між агентами використовуються сокети та мережевий протокол TCP.

2. АНАЛІЗ ПРОГРАМНИХ АГЕНТІВ РЕЄСТРА МУЛЬТИАГЕНТНОЇ СИСТЕМИ

На сьогодні мультиагентні системи є одним з найважливіших напрямків досліджень та розробок в області інформаційних технологій та штучного інтелекту. Мультиагентна система складається з декількох взаємодіючих програмних компонентів – агентів, які здатні співпрацювати між собою для вирішення проблем, які не залежать від можливостей будь-якого окремого агента.

2.1 Існуючі рішення мультиагентних систем

Таблиця 2.1 Існуючі програмні рішення платформ мультиагентних систем

	Jade	Coguaar	Aglobe
Галузь	Мобільні мережі, WEB, планування та логістика, дослідження технологій агентів	Мобільні мережі, WEB, промислове та військове застосування, розподілені системи	Промислове застосування, моделювання інженерних систем
Технології	Java SE, Java, стандарти FIPA	Java SE, Java ME	Java SE
Розширюваність	+	+	—
Наявність плагінів	+	+	—
Інтеграція	Java EE (JMS, Web...), CORBA	Java EE (JMS, Web...), CORBA	—
Засоби візуалізації	—	—	+

2.2 Висновки до розділу

Наразі існує певна кількість готових рішень наприклад Jade, Coguaar, Aglobe, однак вони не мають весь необхідний функціонал, а також їх складно інтегрувати. До того ж їх вихідний код знаходиться у закритому доступі, тож у разі виникнення непередбачуваних помилок, чи необхідності повного розуміння реалізації потрібно звертатися до відділу підтримки вище згаданих продуктів на що піде невизначена кількість часу. У разі самостійної реалізації системи, розробник має повний контроль над усіма процесами і може обирати інструментарій виходячи з поставлених задач.

3. ЗАСОБИ РОЗРОБКИ

Під час написання проекту використано наступні методології та інструментарії розробки.

3.1 Середовище Visual Studio 19

Microsoft Visual Studio – це програмний продукт компанії Microsoft, що надає можливість створювати застосунки, що працюють на платформах .NET[3], Linux, MacOS, Android, хмарних платформах Azure та Amazon Web Services та багатьох інших. Особливість платформи .NET полягає в широкому виборі реалізованих бібліотек, які доступні різними мовами програмування та дозволяють швидко вирішувати тривіальні задачі. При цьому сервіси реалізовано у вигляді проміжного коду, який не залежить від базової архітектури.

Microsoft Visual Studio об'єднує в собі величезну кількість функцій, що дозволяють здійснювати розробки для Windows всіх версій, в тому числі і Windows 10, створювати веб-додатки, Universal Windows Platform та додатки для мобільних пристроїв і хмарних технологій. У Visual Studio реалізовано середовище розробника, завдяки якому процес розробки програм став значно простішим. Microsoft Visual Studio[3] – це оновлене і спрощене програмне середовище, для якого характерна висока продуктивність, причому вона не залежить від особливостей обладнання.

Кожна нова версія програми вміщує в собі новітні інструменти та технології, що дозволяють створювати програми з урахуванням особливостей сучасних платформ. Наприклад, Visual Studio 2019 підтримує більш ранні версії, в тому числі Windows XP, Windows 7, Windows 8 і Windows Server 2003. При цьому розробникам відкрита дорога для створення нових і модернізації вже існуючих додатків, призначених для ранніх версій ОС Windows. Варто зазначити, що в процесі використання підтримуваних

системою версій – вихідні файли, проекти і рішення в програмі будуть працездатними, але вихідний код може потребувати змін.

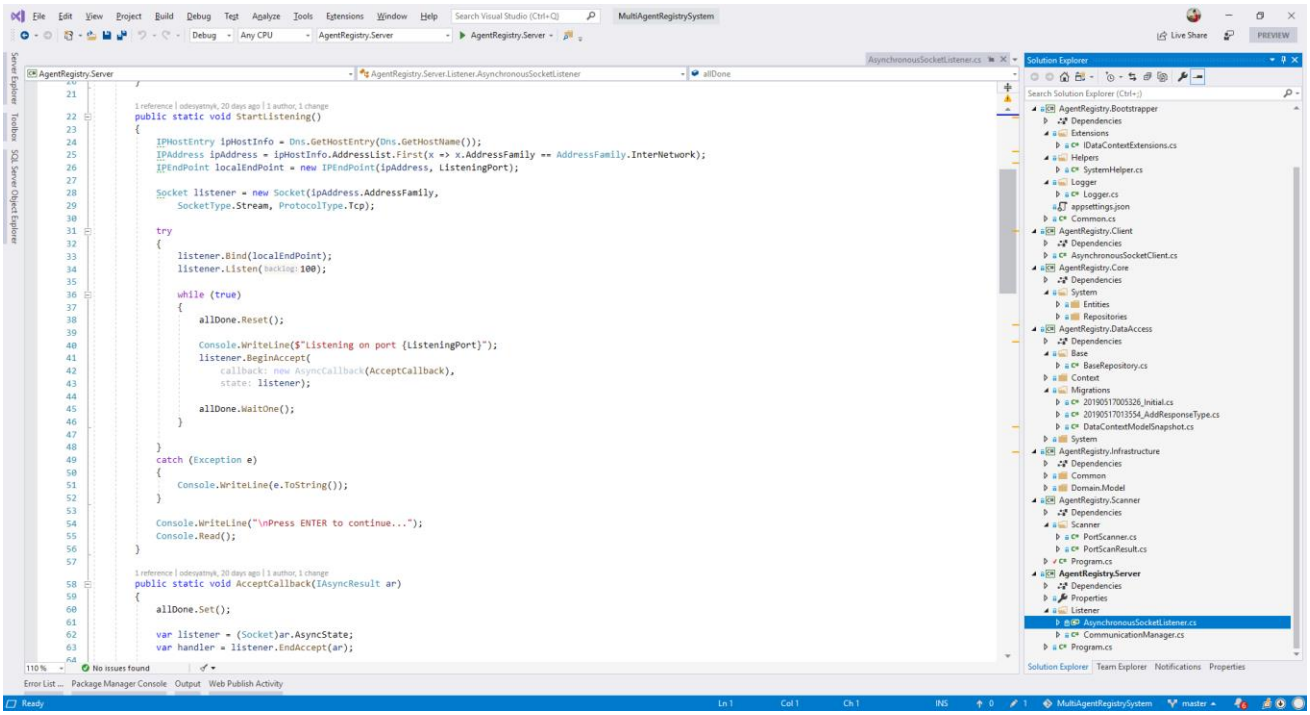

Рисунок 3.1 – Вікно Visual Studio

Visual Studio 2019 (рисунок 1.1) Professional являє собою передову програму, яка дає можливість будь-яким за розміром командам координувати процес проектування та розробки різних програм[4]. Завдяки інструментам гнучкого планування можна впроваджувати методи ітеративної розробки і застосовуватися гнучкі методології в зручному для користувача темпі.

За допомогою засобів моделювання, виявлення і проектування можна максимально повно описати систему, яка дозволить вдало реалізувати конкретну концепцію архітектури. Програмне середовище Visual Studio включає один або декілька з наступних компонентів, в залежності від обраної конфігурації під час встановлення програми:

– Visual F# (починаючи зVisual Studio 2010);
– Visual C#;
– Visual C++;

– Visual J#;
– Visual Studio Debugger;
– Visual Basic .NET;
– Visual Basic.

Архітектура програми Visual Studio підтримує можливість використання розширень (Extensions)[4] – модулів від сторонніх розробників, які дозволяють розширювати можливості середовища розробки. При цьому розробникам відкрита дорога для створення нових і модернізації вже існуючих додатків, призначених для ранніх версій ОС Windows.

Visual Studio 2019 підтримує .NET Core версії 2.2, і дозволяє підтримувати онлайн сервіси ASP .NET. Також підтримує усі версії SQL Server до 2017 року. Було надано можливості для розробки 64 – бітних та ARM застосунків.

3.2 Система управління базами даних MS SQL SERVER

Microsoft SQL Server – система управління базами даних (СУБД), розроблена корпорацією Microsoft. З виходом версії для Windows NT у 1994 році і з тих пір Microsoft підтримує обслуговування продукту[5].

Як база даних, це програмний продукт, основна функція якого полягає в тому, щоб зберігати та отримувати дані, що запитуються іншими програмами, незалежно від того, працюють вони на одному комп'ютері чи через мережу (включаючи Інтернет).

Існує, принаймні, десяток різних видань Microsoft SQL Server, призначених для різних аудиторій і для різних робочих навантажень (від невеликих додатків, які зберігають і отримують дані на одному комп'ютері до мільйонів користувачів і комп'ютерів, які мають доступ до великих обсягів даних одночасно).

Основними мовами запитів є Transact-SQL (T-SQL) і ANSI SQL.

За підтримки Microsoft, протягом багатьох років, вона є однією з провідних реляційних СУБД на ринку. Розподілений, у різних виданнях, та з декількома інтегрованими інструментами, сервер здатний задовольнити вимоги найпростіших задач до найскладніших сценаріїв, які стосуються великих обсягів даних.

SQL Server – реляційна база даних. Це говорить про те, що дані зберігаються у табличному вигляді (рядки і стовпці).

Це дозволяє створювати пов'язані таблиці, уникаючи необхідності зберігати надлишкові дані в декількох місцях в базі даних. Реляційна модель також забезпечує цілісність посилань та інші обмеження цілісності для підтримки узгодженості даних. SQL Server підтримує транзакції, а також принципи атомарності, послідовності, ізоляції і довговічності.

Всі операції, які можна виконувати в SQL Server, передаються йому у форматі, визначеним Microsoft, який називається TDS (табличний потік даних)[6]. TDS – це протокол прикладного рівня, який використовується для передачі даних між сервером баз даних і клієнтом. Спочатку розроблений компанією Sybase Inc. для ядра СУБД Sybase SQL Server у 1984 році а згодом і Microsoft на Microsoft SQL Serve. Пакети TDS можуть бути інкапсульовані в інші фізичні транспортно-залежні протоколи, включаючи TCP/IP. Крім того, API SQL Server також доступний для веб-сервісів.

Мова T-SQL (Transact-SQL) є основним засобом програмування і управління SQL Server. Вона реалізує операції, які можуть виконуватися на SQL Server, включаючи створення і зміну схем бази даних, введення і редагування даних, а також моніторинг і управління самим сервером.

Клієнтські програми, які використовують дані, або керують сервером, використовують функціональні можливості SQL Server, посилаючи запити мовою T-SQL, які обробляються сервером, повертаючи результати (або помилки) клієнтському додатку. SQL

Функціональні можливості управління здійснюються завдяки системно визначеним збереженим процедурам, які можуть бути викликані з T-SQL запитів для виконання операції керування. Також можна створювати пов'язані сервери за допомогою T-SQL[7]. Пов'язані сервери дозволяють одному запиту обробляти операції, що виконуються на декількох серверах.

Програма Microsoft Visual Studio містить вбудовану підтримку Microsoft SQL Server. Може використовуватися для написання та налагодження коду, який буде виконуватися за допомогою SQL CLR. Вона також включає конструктор даних, який можна використовувати для створення, перегляду або графічного редагування схем бази даних. Запити можна створювати візуально або за допомогою коду. SSMS (SQL Server Management Studio)[8], також забезпечує intellisense (технологія автодоповнення Microsoft) для запитів SQL.

Поточна версія Microsoft SQL Server 2017, випущена 2 жовтня 2017 року. Компоненти SQL Server 2017:

- Двигун бази даних;
- Служби аналізу;
- Служби звітності;
- Інтеграційні послуги;
- Основні послуги даних;
- Послуги з машинного навчання (у базі даних);
- Сервер машинного навчання (автономний).

3.3 Платформа .NET Core

.NET Core – це платформа для розробки з відкритим вихідним кодом, керована спільнотою Microsoft і .NET у GitHub[9]. Це крос-платформний продукт, який підтримує

Windows, MacOS і Linux і може бути використаний для створення додатків для різних пристроїв, хмарних сервісів та IoT.

Платформу .NET Core розроблено як дуже подібний, але в той же час унікальний у порівнянні з іншими продуктами .NET. Розроблена, з метою забезпечення широкої пристосованості до нових платформ і навантажень. Вона може розгортатися під керуванням різних операційних систем та архітектур процесора.

Продукт розділений на кілька частин так, що вони можуть бути адаптовані до нових платформ в різний час. Бібліотеки середовища виконання і специфічні щодо платформи бібліотеки повинні бути передані як унікальні елементи[10]. Натомість бібліотеки незалежні від платформи мають працювати самостійно на всіх платформах. З метою підвищення ефективності розробки, під час програмування для платформи .NET Core рекомендується вживати якомога менше специфічного щодо платформи коду, віддаючи перевагу незалежному коду C #, у цьому разі алгоритм або API можуть бути повністю або частково реалізовані таким чином.

Під час розробки для кількох операційних систем доступні окремі реалізації та умовна компіляція, з великою перевагою останньої опції.

Більша частина коду CoreFX[11] є незалежним від платформи і спільно використовується між усіма платформами. Незалежний від платформи код може бути реалізований як окрема портативна збірка, що використовується на всіх платформах.

Платформа .NET була вперше оголошена компанією Microsoft в 2000 році і з того часу значно розвинулася. Технологія .NET Framework була основною реалізацією .NET протягом майже двох десятиліть.

Порівняння із .NET Framework:

– .NET Core не підтримує всі шаблони додатків .NET Framework. Зокрема, він не підтримує ASP.NET і ASP.NET MVC Web Forms, але підтримує ASP.NET Core MVC[12]. Було оголошено, що .NET Core 3 буде підтримувати WPF і Windows Forms;

– .NET Core містить велику підмножину бібліотеки базових класів .NET Framework з різними функціями. Наприклад, імена збірок є різними і відкриті члени класів та їх назви можуть відрізнятися в порівнянні з .NET Framework. Через ці відмінності під час переносу проектів на .NET Core в деяких випадках необхідно внести зміни до вихідного коду програми;

– З метою отримання більш простої реалізації та моделі програмування .NET Core реалізує підмножину систем .NET Framework,. Наприклад, захист доступу до коду не підтримується під час відображення;

– .NET Framework підтримує Windows і Windows Server, в той час як .NET Core також підтримує MacOS і Linux[13];

– .NET Core має відкритий вихідний код, тоді як .NET Framework доступний лише частково, і тільки у форматі перегляду коду.

3.4 Мова програмування C#

Всю логіку процесів системи реалізовано об’єктно-орієнтованою мовою програмування високого рівня C#. Вибір зумовлений тим, що мова C# має велику кількість готових бібліотек класів, що були створені для вирішення широкого спектру прикладних задач. Іншою перевагою C# є швидке налагодження взаємодії між різними технологіями Microsoft.

Окрім взаємодії з іншими технологіями Microsoft, сильною стороною цієї мови є також висока надійність роботи, адже C# – об’єктно-орієнтована мова високого рівня з жорсткою типізацію[14]. Для запобігання неоднозначності та спрощення розуміння коду з мови було виключено множинне наслідування. Замість цього, було введено поняття інтерфейсу, що являє собою, так званий контракт для класів, що будуть його реалізувати. Іншим чинником високої надійності цієї мови програмування є система винятків та обробки помилок, які були поділенні на два види:

– виняткові ситуації, що можуть виникнути під час роботи програми, які обов’язково розробник має обробити, наприклад: користувач увів у поле у якому допустимі символи, або програма намагається відкрити файл на системі, якого не існує;

– ситуації, коли програма зустрічається з неочікуваними труднощами, наприклад: операція над елементами масиву поза його межами, або переповнення пам’яті через допущену помилку програмістом.

Для запобігання останнім ситуаціям, в обраній мові програмування є вбудований сервіс керування пам’яттю[15], що звільняє з оперативної пам’яті дані, які не використовуються. Таким чином, програміст вирішує, коли створювати об’єкт, а віртуальна машина відповідає за звільнення пам’яті після того, як об’єкт стає непотрібним, тобто, немає посилань на нього, збирач сміття автоматично прибирає цей об’єкт із пам’яті.

Об’єктно-орієнтований підхід до написання коду дозволяє оперувати поняттями, що зустрічаються в реальному житті з певною долею абстракції. Парадигма ООП[16] наділила мову такими властивостями як масштабованість, що дає змогу неодноразово розширювати розроблену систему. Розширюваність системи полягає в тому, що в систему можна додавати нові компоненти, без зміни вже існуючих. Іншою перевагою цього підходу є багаторазове використання написаного коду, що значно скорочує його кількість.

Наразі C# являє собою одну з найрозповсюдженіших мов програмування, адже може застосовуватися для вирішення більшості задач.

3.5 Фреймворк Entity Framework Core

Фреймворк Entity Framework Core (EF Core) – остання версія Microsoft Entity Framework[17]. Він розроблений у форматі легкої та розширюваної технології та підтримувати крос-платформну розробку як частину Microsoft .NET Core.

Фреймворк Entity Framework Core розроблений таким чином, щоб його було простіше використовувати і покращити продуктивність в порівнянні з попередніми версіями Entity Framework.

EF Core – це об'єктно–реляційне відображення (ORM). Об'єктно–реляційне відображення – це техніка, яка дозволяє розробникам працювати з даними об'єктно-орієнтованим шляхом, виконуючи роботу, необхідну для відображення об'єктів, визначених у мові програмування та даних, що зберігаються в реляційних джерелах даних[18].

Більшість фреймворків розробки включають бібліотеки, які дозволяють доступ до даних з реляційних баз даних через структури даних, подібні до наборів записів.

Слабо типізований підхід до доступу до даних може спричиняти помилки. Проблеми зазвичай виникають через неправильне введення імені стовпця або з того факту, що ім'я стовпця було змінено в базі даних або шляхом зміни порядку, в якому поля вказуються в операторі SQL без внесення відповідних змін до коду програми[19]. Аналогічно, перетворення типів даних можуть бути невдалими. Код буде все ще компілюватися, але будуть виникати помилки під час виконання програми. Як результат, професійні розробники віддають перевагу роботі з даними в строго типізованому вигляді.

Entity Framework Core успадкував багато від своїх попередників, зокрема від Entity Framework 6. В той же час, повинно бути зрозуміло, що EF Core не є новою версією порівняно з EF 6, а абсолютно іншою технологією, тому EF Core використовує власну систему версій, хоча в цілому вони мають однакові принципи роботи. Поточна версія 2.0 була випущена в серпні 2017 року. І технологія продовжує розвиватися. Те, що вже є в EF Core і що тільки планується додати, можна побачити в дорожній карті продукту на його сторінці GitHub[20].

Як технологія доступу до даних, Entity Framework Core може використовуватися на різних платформах стеку .NET Core. Це стандартні платформи, такі як Windows

Forms, консольні програми, WPF, ASP.NET 4.6 / 4.7. Це нові технології, такі як UWP і ASP.NET Core. У той же час, крос-платформний характер EF Core дозволяє використовувати його не тільки для операційних систем Windows, але і для Linux і Mac OS X[21].

Ключовою концепцією EF Core є поняття сутності. Сутність визначає набір даних, пов'язаних з певним об'єктом. Тому ця технологія передбачає роботу не з таблицями, а з об'єктами та їх колекціями.

Будь-яка сутність, як і будь-який об'єкт у реальному світі, має ряд властивостей. Наприклад, якщо сутність описує людину, ми можемо розрізняти такі властивості, як ім'я, прізвище, зріст, вік. Властивості не обов'язково являють собою прості типи даних, як, наприклад, число або рядок символів, але можуть також представляти більш складні структури даних. І кожен об'єкт може мати одне або більше властивостей, які будуть відрізняти цю сутність від інших і однозначно визначатиме її. Ці властивості називаються ключами.

У цьому випадку об'єкти можуть бути з'єднані відношеннями один-до-багатьох, один-до-один і багато-до-багатьох, так само, як у реальній базі даних, зв'язок відбувається через зовнішні ключі[22].

Відмінною особливістю Entity Framework Core, як і технології ORM, є використання запитів LINQ (Language Integrated Query)[23] для отримання даних з бази даних. За допомогою LINQ ми можемо будувати різні запити для створення вибірки об'єктів, включаючи різні пов'язані асоціативні посилання. Під час виконання запиту Entity Framework переводить вирази LINQ у зрозумілі вирази для даної СУБД (зазвичай у виразах SQL).

3.6 Програмний інтерфейс Socket API

Мережевий сокет (Socket) є внутрішньою кінцевою точкою для відправлення або отримання даних у межах вузла комп'ютерної мережі[24]. Зокрема, це подання цієї кінцевої точки в мережевому програмному забезпеченні, наприклад запис у таблиці і є формою системного ресурсу.

Процес може посилатися на сокет, використовуючи дескриптор сокета. Процес спочатку вимагає, щоб стек протоколу створив сокет, і стек повертає дескриптор процесу, щоб він міг ідентифікувати сокет. Потім процес передає дескриптор в стек протоколу, коли він хоче передавати або отримувати дані за допомогою цього сокета.

Сокет зазвичай посилається на інший сокет в мережі IP, особливо для протоколу управління передачею (TCP), який є протоколом один-до-одного. Мається на увазі, що сокети пов'язані з IP-адресою і номером порту для локального вузла, і є відповідна адреса сокету на зовнішньому вузлі, у свою чергу. асоційований сокет, що використовується іншим процесом. Асоціація сокета з адресою сокета називається прив'язка.

В той час як локальний процес може обмінюватися даними з зовнішнім процесом, надсилаючи або приймаючи дані до або з зовнішньої адреси сокету, він не має доступу до самого зовнішнього сокету, і не може використовувати дескриптор зовнішнього сокета, оскільки обидва є внутрішніми для зовнішнього вузла.

Стек протоколів, зазвичай наданий операційною системою, являє собою набір послуг, які дозволяють процесам здійснювати зв'язок через мережу з використанням протоколів, реалізованих стеком. API (Application Programming Interface), який програми використовують для зв'язку з стеком протоколів, використовуючи мережеві сокети, називається сокетним API.

У інтернет-протоколі TCP адреса сокету є комбінацією IP-адреси і номера порту. Сервер TCP може обслуговувати декілька клієнтів одночасно, створюючи дочірній

процес для кожного клієнта і встановлюючи з'єднання TCP між дочірнім процесом і клієнтом. Для кожного з'єднання створюються унікальні виділені сокети[21].

Сервер може створити декілька одночасно встановлених TCP-сокетів з однаковим номером локального порту та локальною IP-адресою, кожен з яких пов'язаний з власним процесом, який обслуговує свій клієнтський процес.

3.7 Висновки до розділу

Під час розробки програмного продукту були використані такі технології та середовища як Microsoft Visual Studio 2019, SQL Server Management Studio, об’єктно–орієнтована мова програмування C#, програмний інтерфейс Socket API на базі платформи .NET Core. Вибір цих технологій обумовлено тим що вони є продуктом Microsoft тому їх легко поєднати в єдину систему та мають високу продуктивність взаємодії.

Цільовою платформою обрано .NET Core, оскільки ця додатки розроблені на її базі можливо розгорнути у більшості популярних операційних системах. У якості об’єктно-реляційної технології відображення обрано Entity Framework Core.

4. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Система складається із 8 проектів, які побудовані за гексагональною архітектурою з використанням техніки DDD (Domain Driven Design).

4.1 Архітектура системи

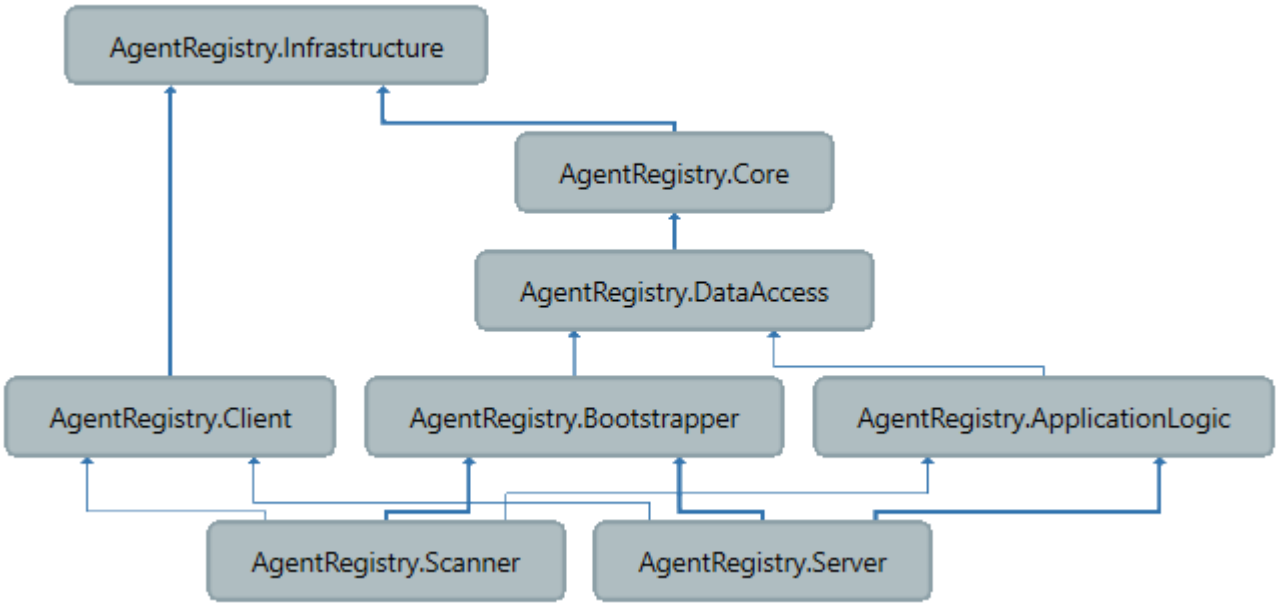

Рисунок 4.1 – структура проекту

В основі структури проекту (рисунок 4.1) полягає гексагональна архітектура.

Гексагональна архітектура дозволяє взаємодіяти з додатком як користувачу, так і програмам, автоматичним тестам, скриптам пакетної обробки. Також дозволяє розробляти і тестувати програми без будь-яких додаткових пристроїв або баз даних.

Існують дві основні області – внутрішня і зовнішня. Зовнішня – дозволяє клієнтам вводити дані, зберігати результати роботи програми або посилати їх куди-небудь. Внутрішня – містить модель предметної області і набір служб додатку, що реалізують

універсальний програмний інтерфейс для роботи з цією доменної моделлю. Гексагональна архітектура називається також «порти і адаптери».

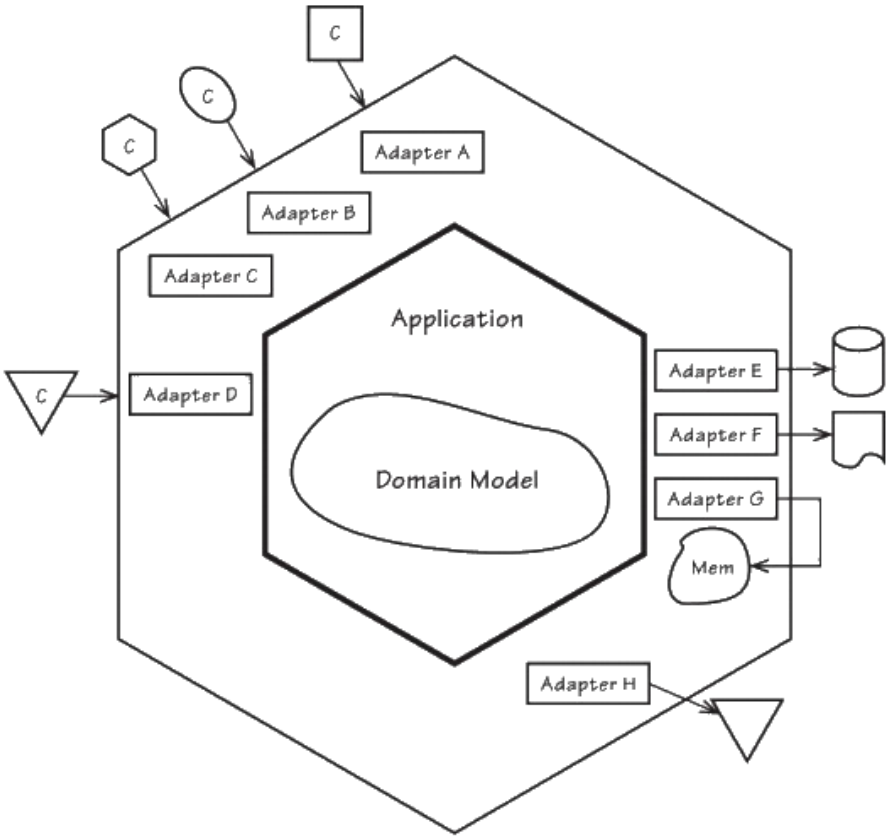

Рисунок 4.2 – модель гексагональної архітектури

Кожна грань являє собою «порт» доступу до додатку або ж його зв'язок із зовнішнім світом (рисунок 4.2). Порт може бути представлений як провідник вхідних запитів (або даних) до нашого додатку.

Наприклад, через TCP-порт приходять запити, які конвертуються в команду для програми. Схожим чином можуть діяти різні менеджери черг. Все це є лише портами через які є можливість попросити додаток зробити якісь дії. Ці межі складають безліч «вхідних портів». Інші порти можуть бути використані для доступу до даних з боку додатка, наприклад порт бази даних.

Адаптер трансформує протоколи введення в інформацію, сумісну з внутрішнім інтерфейсом програми. Для кожного з зовнішніх типів даних існує окремий адаптер.

Будь-яка кількість клієнтів будь-якого типу може звертатися до різних портів, але при цьому кожен адаптер делегує їх додатку, використовуючи один і той же внутрішній програмний інтерфейс.

Програмний інтерфейс додатку публікується як безліч служб. В даному випадку служби програми є безпосередніми клієнтами моделі предметної області.

При правильному проектуванні, внутрішній шестикутник – додаток і модель предметної області – ніяк не вплине на зовнішні частини.

Це створює чіткі межі, всередині яких реалізовано кожен сценарій використання. Крім довільної кількості клієнтів, адаптер може підтримувати численні автоматизовані тести і реальних клієнтів, а також механізми зберігання, розсилки повідомлень та інші засоби виведення.

4.2 Опис структури проекту

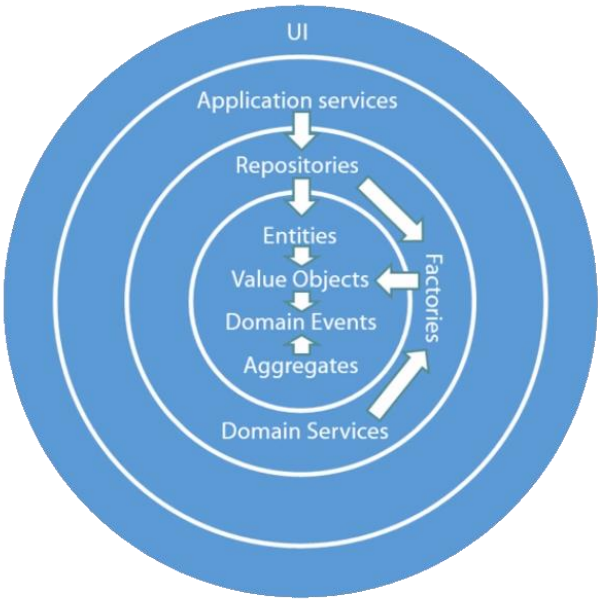

Рисунок 4.3 – класична модель проекту з використанням DDD

За основу побудови логічного рівня програмного забезпечення обрано дизайн, керований доменом (DDD – Domain Driven Design) (рисунок 4.3).

Цей шаблон спрямований на полегшення створення складних додатків шляхом підключення пов'язаних частин програмного забезпечення до моделі що постійно розвивається. Концепція DDD зосереджується на трьох основних принципах:

– зосередження на логікі предметної області;

– базові комплексні конструкції на моделях домену;

– постійна співпраця з експертами доменів, задля поліпшення моделі програми та вирішення будь-яких нових проблем, пов'язаних із предметною областю.

Дизайн, що керується доменом, також визначає ряд концепцій високого рівня, які можна використовувати разом з іншими, для створення та модифікації моделей доменів.

Сутність – об'єкт, який ідентифікується відповідно до послідовності на відміну від традиційних об'єктів, які визначаються їх атрибутами.

Value Object – незмінний об'єкт, який має атрибути, але не має окремої ідентичності.

Подія – об'єкт, який використовується для запису дискретної події, пов'язаної з діяльністю моделі в системі.

Агрегат – кластер об'єктів і об'єкти значення з певними межами навколо групи. Замість того, щоб дозволити кожному окремому об'єкту або об'єкту Value Object виконувати всі дії самостійно, колективній сукупності елементів присвоюється єдиний сукупний кореневий елемент. Тепер зовнішні об'єкти більше не мають прямого доступу до кожного окремого об'єкта або об'єкта Value Object в межах сукупності, а мають доступ до єдиного сукупного кореневого елемента і використовують його для передачі інструкцій групі в цілому.

Сервіс – це операція або форма бізнес-логіки, яка природно не вписується в сферу об'єктів. Іншими словами, якщо певна функціональність повинна існувати, але вона не може бути пов'язана з об'єктом сутності або Value Object, це, ймовірно, сервіс.

Репозиторій – це служба, яка використовує глобальний інтерфейс для забезпечення доступу до всіх об'єктів і, які знаходяться в певній сукупній колекції.

Методи повинні бути визначені для створення, модифікації та видалення об'єктів у межах агрегату. Однак, використовуючи цю службу для створення запитів даних, мета полягає в тому, щоб виключити таку можливість запиту даних з бізнес-логіки об'єктних моделей.

Дизайн, що керується доменом, також сильно підкреслює все більш популярну практику безперервної інтеграції, яка змушує всю команду розробників використовувати один спільний репозиторій коду. Автоматичний процес, що відбувається наприкінці робочого дня, перевіряє цілісність всієї бази коду, запущених автоматизованих тестових одиниць, регресійних тестів і тому подібного, щоб швидко виявити будь-які потенційні проблеми, які могли бути додані з останніми змінами коду.

Серед переваг дизайну, керованого доменом можна виділити легку комунікацію. Створення загальної та повсюдної мови, пов'язаної з моделлю домену проекту, буде корисним протягом усього життєвого циклу програмного продукту. Як правило, DDD вимагає менше технічного жаргону при обговоренні аспектів застосування, оскільки повсюдна мова, встановлена на ранній стадії, ймовірно, визначатиме більш прості терміни для позначення технічних аспектів.

Оскільки DDD настільки сильно базується на концепціях об'єктно-орієнтованого аналізу та проектування, майже все, що входить в модель домену, буде засноване на об'єкті і, отже, буде досить модульним та інкапсульованим.

4.3 Концептуальна модель бази даних

Концептуальна модель (рисунок 4.4) — це відображення предметної області, для якої розробляється база даних. Нинішні серверні системи управління базами даних побудовані на базі реляційної моделі організації даних.

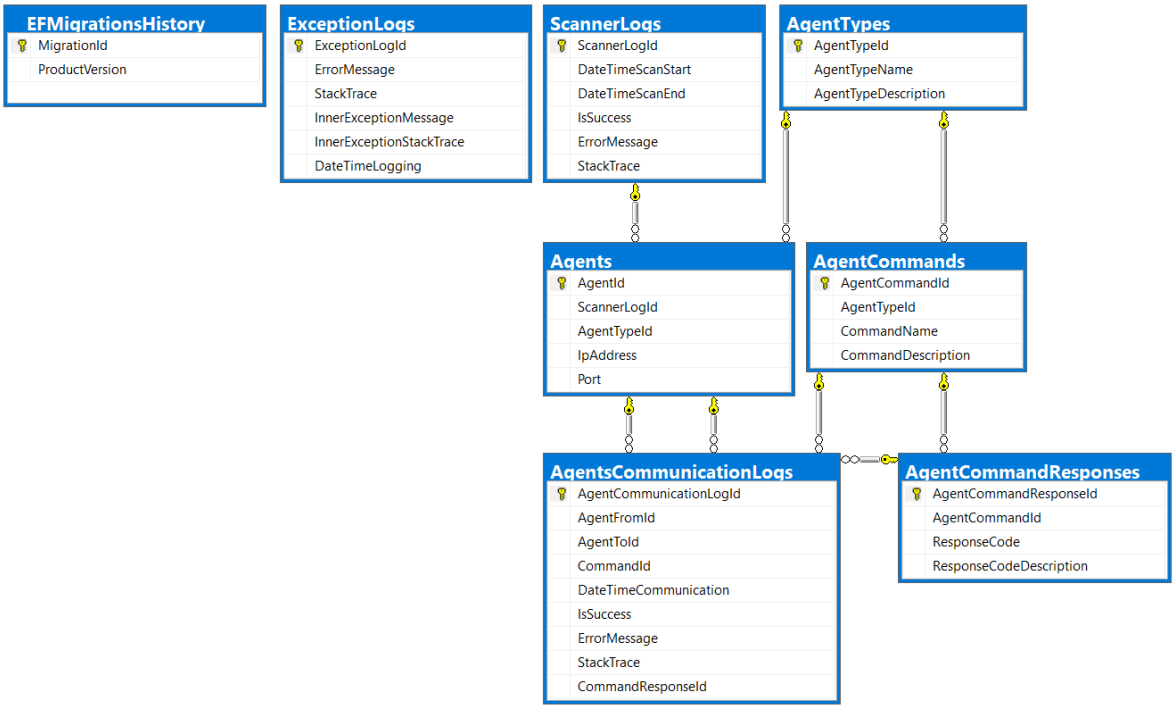

Рисунок 4.4 – Концептуальна модель бази даних

Модель бази даних даної предметної області складається із восьми таблиць: «EFMigrationHistory», «ExceptionLogs», «ScannerLogs», «AgentsCommunicationLogs», «AgentTypes», «Agents», «AgentCommands», «AgentCommandResponses». Всі таблиці, окрім «EFMigrationHistory» та «ExceptionLogs», зв’язані між собою за допомогою зовнішніх ключів, що забезпечує цілісність даних.

Таблиця «EFMigrationHistory» (таблиця 4.1) містить інформацію про історію змін схеми бази даних, а саме MigrationId – унікальний ідентифікатор зміни схеми бази даних, ProductVersion – унікальний ідентифікатор схеми бази данихзміни.

Таблиця 4.1 Поля таблиці «EFMigrationHistory»

Поле	**Тип**	**Опис**
MigrationId	nvarchar(150)	Ідентифікатор запису
ProductVersion	nvarchar(32)	Ідентифікатор версії схеми бази даних

Таблиця «ExceptionLogs» (таблиця 4.2) містить інформацію про історію помилок які відбувалися під час роботи системи, а саме ExceptionLogId – унікальний

ідентифікатор помилки, ErrorMessage – короткий опис помилки, StackTrace – розгорнутий стек коду, який є інформацією про послідовність виконаних операцій які і призвели до помилки, InnerExceptionMessage – короткий опис вкладеної помилки, якщо вона присутня, InnerExceptionStackTrace – розгорнутий стек виконаних операцій, які призвели до виникнення вкладеної помилки, DateTimeLogging – відмітка часу коли відбулася помилка

Таблиця 4.2 Поля таблиці «ExceptionLogs»

Поле	**Тип**	**Опис**
ExceptionLogId	int	Ідентифікатор помилки
ErrorMessage	nvarchar(max)	Короткий опис помилки
StackTrace	nvarchar(max)	Стек операцій що спричинили помилку
InnerExceptionMessage	nvarchar(max)	Короткий опис вкладеної помилки
InnerExceptionStackTrace	nvarchar(max)	Стек операцій що спричинили вкладену помилку
DateTimeLogging	datetime	Відмітка часу коли відбулася помилка

Таблиця «ScannerLogs» (таблиця 4.3) – зберігає результати моніторингу мережі на наявність агентів у системі, а саме ScannerLogId – ідентифікатор сесії сканування мережі, DateTimeScanStart – відмітка часу початку сесії сканування мережі, DateTimeScanEnd – відмітка часу закінчення сесії сканування мережі, IsSuccess – прапорець, який ідентифікує чи сесія сканування мережі завершилася вдало, ErrorMessage – у разі невдалого завершення сесії сканування мережі зберігає текст помилки, StackTrace – у разі невдалого завершення сесії сканування мережі, зберігає каскад виконаних операцій які призвели до помилки.

Таблиця 4.3 Поля таблиці «ScannerLogs»

Поле	**Тип**	**Опис**
ScannerLogId	int	Ідентифікатор сесії сканування
DateTimeScanStart	datetime	Відмітка часу початку сесії сканування
DateTimeScanEnd	datetime	Відмітка часу кінця сесії сканування
IsSuccess	bit	Прапорець статусу сесії сканування
ErrorMessage	nvarchar(max)	Короткий опис помилки
StackTrace	nvarchar(max)	Стек операцій що спричинили помилку

Таблиця «AgentsCommunicationLogs» (таблиця 4.4) – зберігає історію комунікації агентів, а саме AgentCommunicationLogId – ідентифікатор виконаної операції взаємодії між агентами, AgentFromId ідентифікатор агента, який ініціював сеанс взаємодії, AgentToId – ідентифікатор цільового агента взаємодії, CommandId – ідентифікатор запитуваної команди на цільовому агенті, DateTimeCommunication – відмітка часу сеансу взаємодії, IsSuccess – прапорець статусу взаємодії, ErrorMessage – опис помилки у разі невдалого завершення сеансу, StackTrace – стек операцій що спричинили помилку, CommandResponseId – ідентифікатор коду відповіді цільового агента, агенту який ініціював сеанс.

Таблиця 4.4 Поля таблиці «AgentsCommunicationLogs»

Поле	**Тип**	**Опис**
AgentCommunicationLogId	int	Ідентифікатор сеансу взаємодії
AgentFromId	int	Ідентифікатор агента ініціатора
AgentToId	int	Ідентифікатор цільового агента
CommandId	int	Ідентифікатор команди
DateTimeCommunication	datetime	Відмітка часу сеансу
IsSuccess	bit	Прапорець статусу сеансу
ErrorMessage	nvarchar(max)	Короткий опис помилки
StackTrace	nvarchar(max)	Стек операцій що спричинили помилку
CommandResponseId	int	Ідентифікатор коду відповіді

Таблиця «AgentTypes» (таблиця 4.5) – зберігає інформацію про всі можливі типи агентів які можуть бути присутні в системі, а саме AgentTypeId – ідентифікатор типу агента, AgentTypeName – назва типу агента, AgentTypeDescription – опис типу агента.

Таблиця 4.5 Поля таблиці «AgentTypes»

Поле	**Тип**	**Опис**
AgentTypeId	int	Ідентифікатор типу агента
AgentTypeName	nvarchar(max)	Назва типу агента
AgentTypeDescription	nvarchar(max)	Опис типу агента

Таблиця «Agents» (таблиця 4.6) – заповняється під час сесії сканування мережі на

доступність агентів, зберігає інформацію про доступних агентів в конкретний момент часу, а саме AgentId – ідентифікатор агента, ScannerLogId – ідентифікатор сесії сканування мережі, під час якої було виявлено агента, AgentTypeId – ідентифікатор типу агента, IpAddress – мережевий IPv4 адрес агента, Port – номер відкритого порту агента через який можна із ним взаємодіяти.

Таблиця 4.6 Поля таблиці «Agents»

Поле	**Тип**	**Опис**
AgentId	int	Ідентифікатор агента
ScannerLogId	int	Ідентифікатор сесії сканування мережі
AgentTypeId	int	Ідентифікатор типу агента
IpAddress	nvarchar(max)	Мережева адреса агента
Port	int	Відкритий порт агента

Таблиця «AgentCommands» (таблиця 4.7) – зберігає інформацію про всі доступні команди всіх типів агентів, а саме AgentCommandId – ідентифікатор команди, AgentTypeId – ідентифікатор типу агента якому належить команда, CommandName – назва команди, CommandDescription – опис команди.

Таблиця 4.7 Поля таблиці «AgentCommands»

Поле	**Тип**	**Опис**
AgentCommandId	int	Ідентифікатор команди
AgentTypeId	int	Ідентифікатор типу агента власника
CommandName	nvarchar(max)	Назва команди
CommandDescription	nvarchar(max)	Опис команди

Таблиця «AgentCommandResponses» (таблиця 4.8) – зберігає дані про всі можливі коди відповідей на команди агентів, а саме AgentCommandResponseId – ідентифікатор коду відповіді, AgentCommandId ідентифікатор команди до якої належить код відповіді, ResponseCode – код відповіді агента на команду, ResponseCodeDescription – опис коду відповіді.

Таблиця 4.8 Поля таблиці «AgentCommands»

Поле	**Тип**	**Опис**
AgentCommandResponseId	int	Ідентифікатор відповіді
AgentCommandId	int	Ідентифікатор команди
ResponseCode	nvarchar(max)	Код відповіді
ResponseCodeDescription	nvarchar(max)	Опис коду відповіді на команду

4.4 Опис класів системи

Проект Infrastructure (рисунок 4.5) містить у собі класи спільні для всіх інших проектів, зокрема класи DTO (Data Transfer Object), базові інтерфейси сутностей,

репозиторіїв, системи логування а також базовий інтерфейс класу контексту доступу до бази даних.

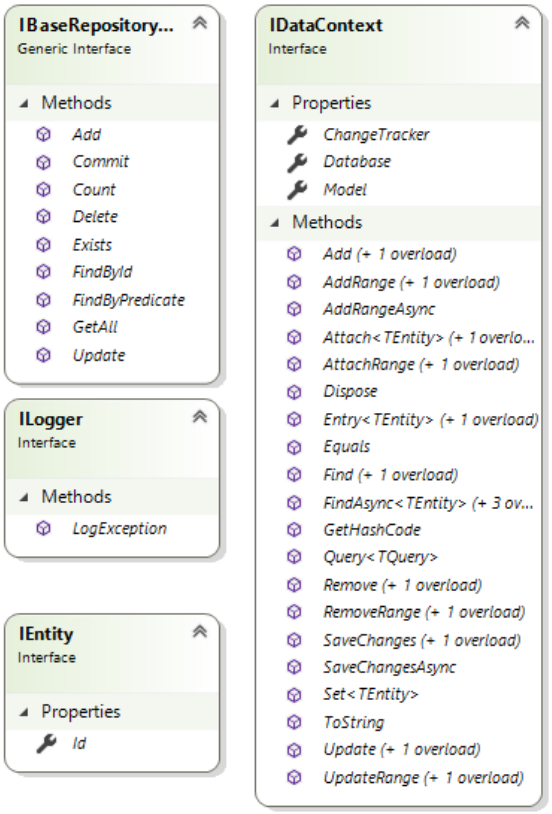

Рисунок 4.5 – Інтерфейси проекту Infrastructure

Інтерфейс IEntity – базовий інтерфейс для всіх домених класів моделей таблиць бази даних. Класи котрі реалізують цей інтерфейс будуть мати обов’язкову властивість – ідентифікатор.

Інтерфейс ILogger – базовий інтерфейс для службового класу який виконує роль реєстратора помилок системи. Реалізація метода LogException має виконувати функцію запису до бази даних помилки з всією необхідною інформацією.

Інтерфейс IRepositoryBase містить набір сигнатур методів, котрі будуть виконувати операції маніпуляції сутностями з базою даних.

Інтерфейс IDataContext базовий інтерфейс класу контексту доступу до бази даних.

Проект Core (рисунок 4.6) містить у собі сутності моделей бази даних, інтерфейси репозиторіїв

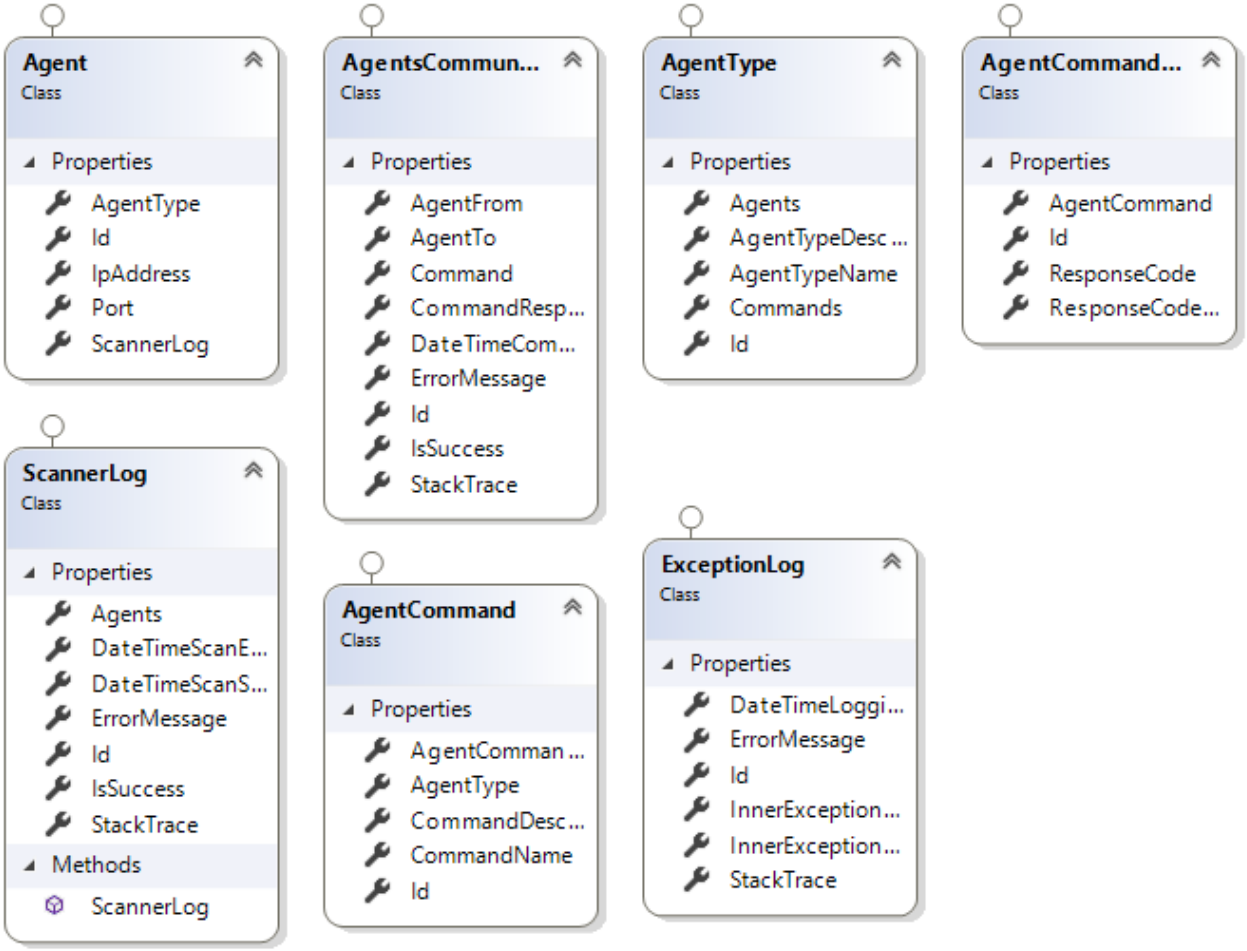

Рисунок 4.6 – Класи проекту Core

Кожен із класів являє собою модель відповідної таблиці в базі даних зі всіма необхідними властивостями.

Проект DataAccess містить у собі реалізації репозиторіїв та контекст бази даних. Призначенням класів цього проекту (рисунок 4.7) є реалізація доступу до бази даних MSSQL. Доступ до бази даних здійснюється за допомогою контексту даних Entity Framework.

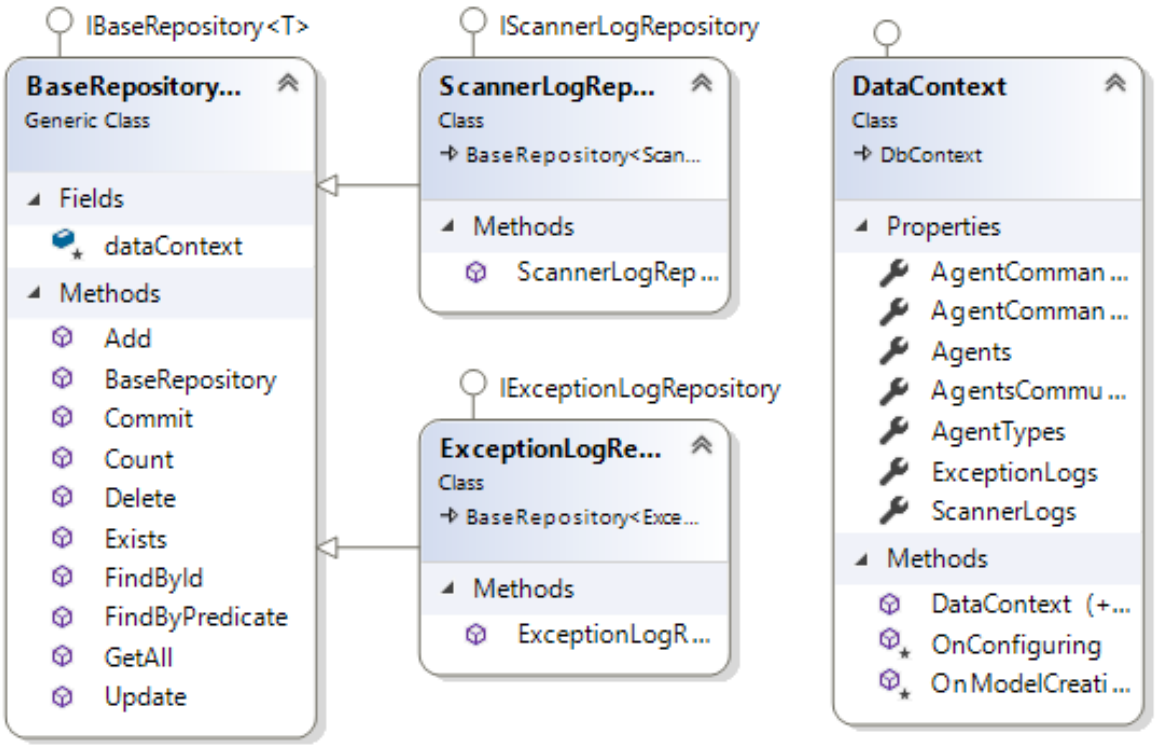

Рисунок 4.7 – Класи проекту DataAccess

Клас RepositoryBase містить реалізацію інтерфейсу IRepositoryBase. Він містить набір методів, які дають можливість брати, видаляти та зберігати дані із бази даних. Метод Delete() дає змогу видалити елемент по його Id. Метод GetAll() дає змогу вибрати всі дані із конкретної таблиці. Метод GetById() повертає елемент із бази даних за його Id. Метод Save() та SaveChanges() дає змогу зберегти елемент та зміни з відповідних елементом до бази даних.

Проект Bootstrapper (рисунок 4.8) містить функціонал спільний для шарів Scanner та Server. Найважливіший метод цього шару Bootstrap викликається в цих шарах для початкової конфігурації сервера та клієнта.

Клас Common містить в собі властивості які відповідають за конфігурацію проектів, а також через для простоти реалізації і впевненості в тому що в системі завжди буде використовуватися єдиний контекст даних цей клас надає доступ до контексту даних за допомогою властивості DataContext.

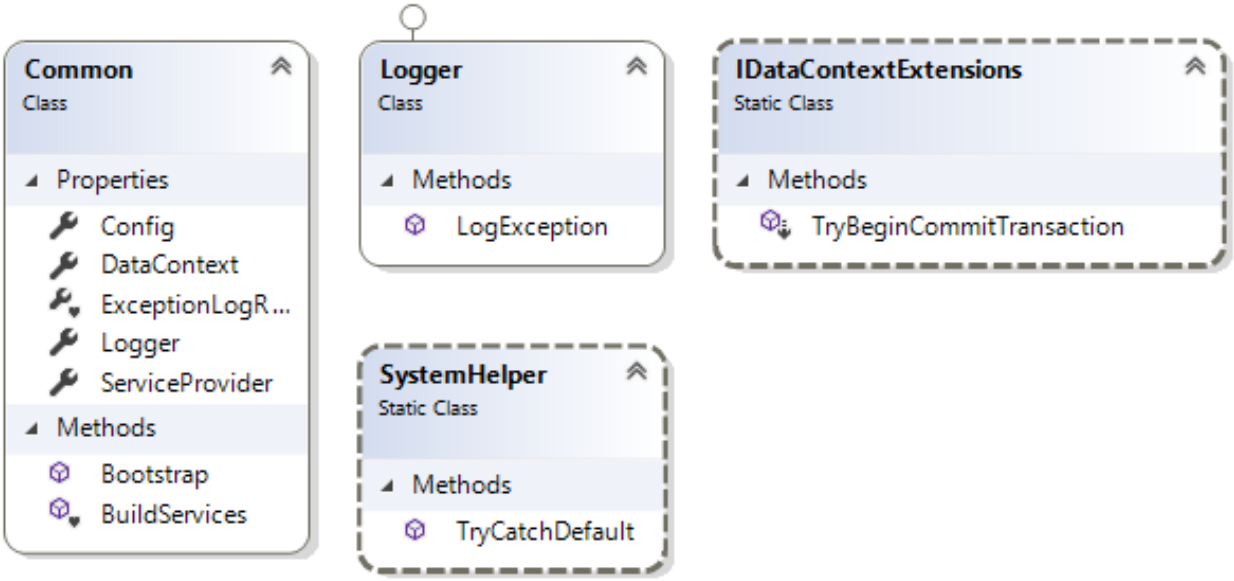

Рисунок 4.8 – Класи проекту Bootstrapper

Клас Logger реалізує інтерфейс ILogger та виконує роль реєстратора помилок системи.

Статичний клас SystemHelper має єдиний метод-обгортку через який можна безпечно здійснювати будь-які операції, та помилки, які виникнуть будуть залоговані.

Клас IDataContextExtensions – клас-розширення основного функціоналу класів, котрі реалізують інтерфейс IDataContext. Метод TryBeginCommitTransaction необхідно використовувати під час роботи із даними бази даних, оскільки за його допомогою можна виконувати операції запису та оновлення даних, а у разі помилки, задля забезпечення цілісності даних, всі внесені зміни відміняться та помилка буде залогована.

Проект ApplicationLogic (рисунок 4.9) містить методи, які виконують головні операції системи, в тому числі взаємодії із базою даних.

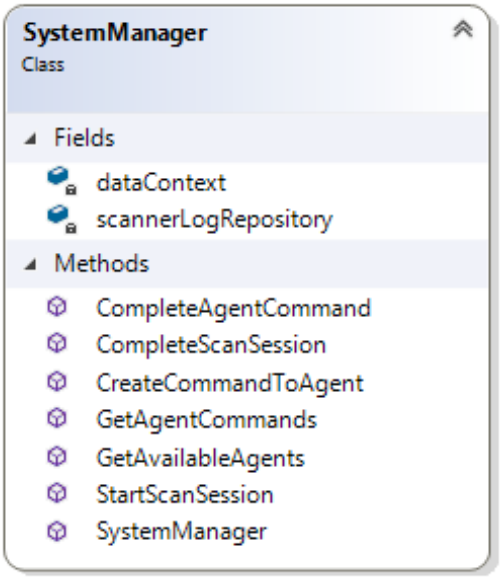

Рисунок 4.9 – Класи проекту ApplicationLogic

Методи StartScanSession та CompleteScanSession служать для управління процесом моніторингу системи пошуку доступних агентів, зокрема записують до бази даних результати сканування.

Методи CreateCommandToAgent та CompleteAgentCommand використовуються для обробки сеансів взаємодії між агентами та зберігають до бази даних всю історію комунікації агентів.

Метод GetAgentCommands повертає список доступних команд за типом агента.

Метод GetAvailableAgents повертає список доступних агентів за результатами останнього вдалого сканування мережі.

Проект Client (рисунок 4.10) містить реалізацію сокета клієнта, котрий використовується для відправки повідомлень. Клас AsynchronousClient, з метою уникнення повторювання коду, винесено в окремий проект оскільки він використовується в проектах Scanner та Server.

Клас AsynchronousClient має лише один публічний метод SendMessage, за допомогою якого можна відправляти повідомлення агентам, знаючи їх IP адресу та номер відкритого порту.

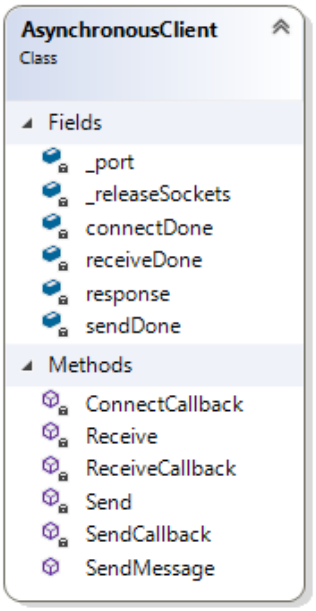

Рисунок 4.10 – Класи проекту Client

Проект Scanner (рисунок 4.11) реалізує сервіс моніторингу мережі на наявність доступних агентів, зчитування їх стану та записує результати сканування до бази даних.

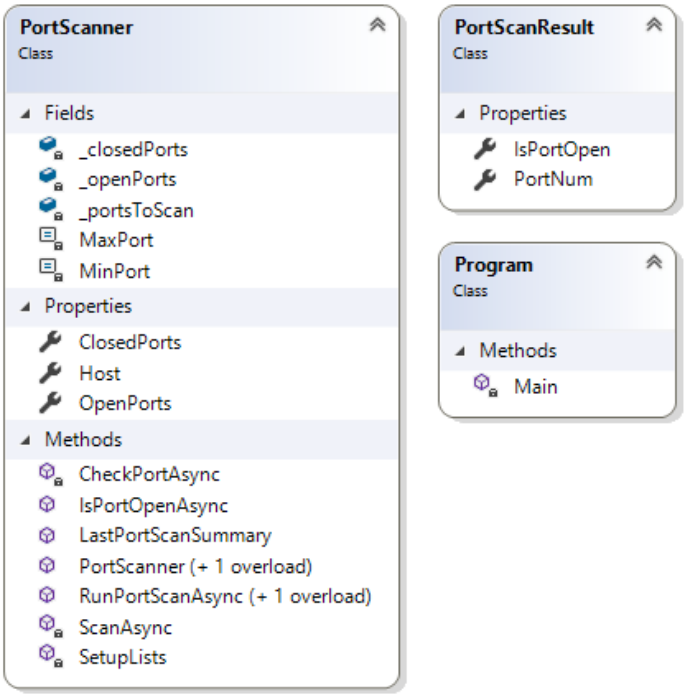

Рисунок 4.11 – Класи проекту Scanner

Клас PortScanResult використовується для структурованої передачі результатів між методами класів.

Клас Program – вхідна точка проекту Scanner, містить сервіс моніторингу мережі, який за заданим графіком проводить сканування доступних агентів та актуалізує дані бази даних згідно з результатами сканування.

Клас PortScanner – реалізує сервіс моніторингу мережі. За заданими параметрами (адреса мережі та діапазон портів) проводить сканування мережі шляхом асинхронної відправки повідомлень на задані мережеві адреси використовуючи клієнтський сокет (реалізація в проекті Client). Процес сканування відбувається шляхом відправки та отримання повідомлення.

Проект Server (рисунок 4.12) – є головним компонентом архітектури реєстру MAC. Являється вхідною точкою додатку, під час запуску вмикає режим очікування запитів та обробляє їх в асинхронному режимі. Веде логування всіх команд агентів а також журнал історії помилок.

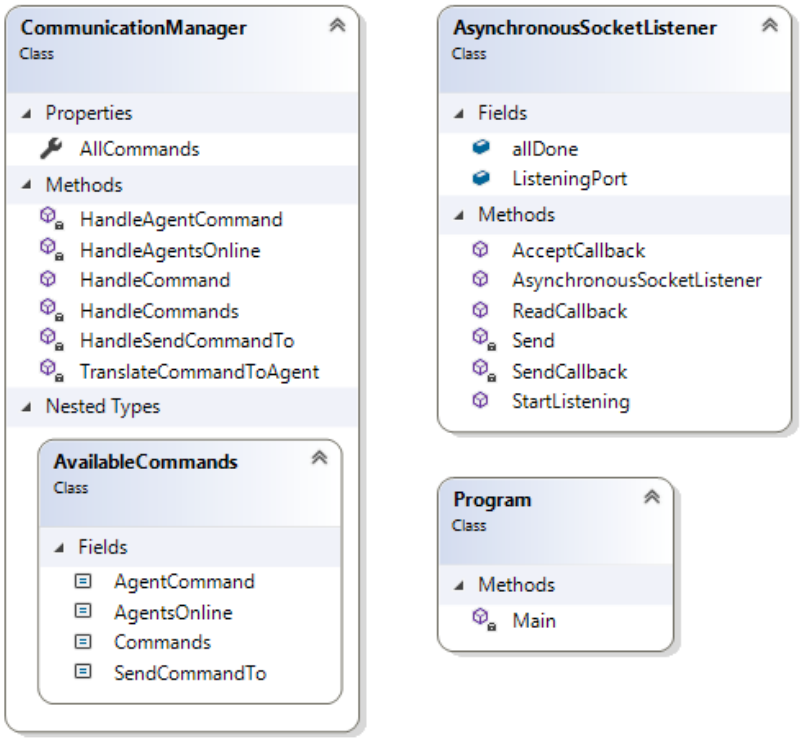

Рисунок 4.12 – Класи проекту Server

Клас Program – вхідна точка проекту Server. Під час запуску, сервер вмикає режим прослуховування заданого порту та чекає на вхідні підключення.

Клас CommunicationManager містить набір методів, які обробляють вхідні запити до сервера.

4.5 Висновки до розділу

Було реалізовано програмний агент реєстру мультиагентної системи, який складається з двох основних компонентів, а саме серверного консольного додатку із двома активними сокетами (клієнт, сервер) та сервіс моніторингу доступних агентів на базі платформи .NET Core.

Функціонал системи полягає в:

– забезпечення стабільності функціонування реєстру MAC;

– зберігання історії виконаних операцій;

– ведення журналу помилок системи;

– забезпечення надійної комунікації агентів MAC за допомогою сервісу «Yellow Pages»;

– проведення моніторингу мережі на предмет доступності агентів.

За основу взято гексагональну архітектуру проектування та технологію розробки додатків DDD (Domain Driven Design).

5. РОБОТА КОРИСТУВАЧА В СИСТЕМІ

Для роботи системи необхідно встановити додаток на пристрій який буде виконувати функцію реєстра MAC і підключити його до тієї ж мережі в котрій знаходяться агенти.

5.1 Інсталяція та системні вимоги

Для роботи користувача з розробленою програмною системою користувачу потрібні мінімальні потужності апаратного забезпечення (таблиця 5.1).

Таблиця 5.1. Вимоги до апаратного забезпечення

Пристрій	Характеристика
Процесор	Intel ® Core ™ 2 / 2 Duo / Pentium ® / Celeron ® / Xeon™ / i3 / i5 / i7 чи AMD 6 / Turion ™ / Athlon ™ / Duron ™ / Sempron ™ з тактовою частотою не нижче 1.5 GHz.
Оперативна пам’ять (RAM − Random Access Memory)	Рекомендовано не менше 4 RAM

Підтримувані апаратні архітектури:

– win-x86
– win/linux/osx-x64
– win/linux-arm

5.2 Сценарії роботи користувача із системою

Користувачами системи являються агенти мультиагентної системи. Інтерфейс користувача реалізовано за шаблоном консольного додатку, оскільки користувачі напряму із ним не взаємодіють а спілкування між агентами здійснюється за допомогою повідомлень на базі сокетів по каналу TCP у форматі символьних рядків за заздалегідь визначеним синтаксисом. Вхідною інформацією є рядок символів який транслюється у команду зрозумілу для сервера.

```
Waiting for a connection...
1Socket connected to 192.168.1.112:11000
Sent 13 bytes to server.
Response received : Available commands:
commands
agents_online
agent_commands
send_command_to<EOF>
2Socket connected to 192.168.1.112:11000
Sent 18 bytes to server.
Response received :
Available agents:
Agent Id - 1007; Agent Type - temperature; IP Address - 192.168.1.112; Port - 11002
Agent Id - 1006; Agent Type - ventilation; IP Address - 192.168.1.112; Port - 11001
<EOF>
```

Рисунок 5.1 – вивід консолі агента

На наведених скріншотах подано приклад взаємодії агента із сервером реєстру мультиагентної системи. Агент спочатку виконує підключення до сервера (рисунок 5.1), про що свідчить відповідне повідомлення у консолі агента, після переконання в тому що з’єднання встановлено коректно, агент відправляє повідомлення у вигляді рядка символів та очікує на відповідь. У наведеному прикладі взаємодії залучається лише один агент та сервер.

У вікні консолі сервера (рисунок 5.2) виводяться повідомлення про вхідні підключення та тексти повідомлень від агентів. У разі успішного виконання команди, сервер повертає повідомлення із результатом запиту.

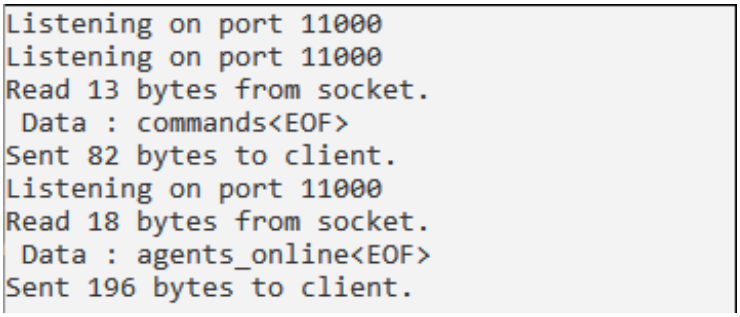

Рисунок 5.2 – вивід консолі сервера

Під час надсилання на сервер команди для виконання на іншому агенті (рисунок 5.3) повторюється процедура описана у першому прикладі взаємодії.

Рисунок 5.3 – вивід консолі першого агента

Однак після отримання повідомлення сервером (рисунок 5.4), він ініціює підключення до цільового агента та відправляє команду до нього, пересвідчившись заздалегідь у правильності виконаного підключення.

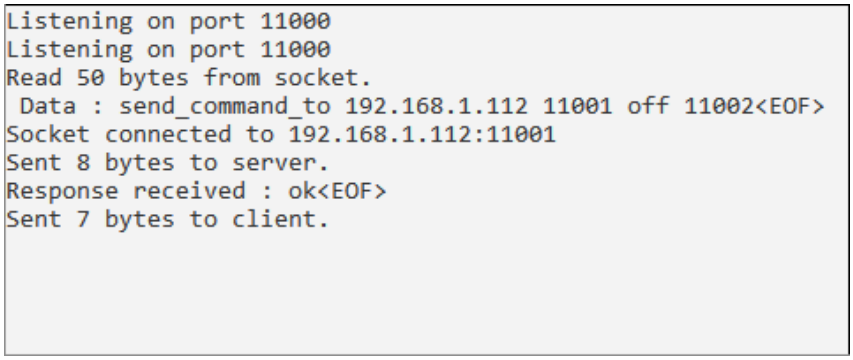

Рисунок 5.4 – вивід консолі сервера під час взаємодії двох агентів

Сервер очікує на відповідь, та відправляє результат виконання команди (рисунок 5.5). до агента ініціатора сеансу взаємодії

```
Waiting for a connection...
Waiting for a connection...
Read 8 bytes from socket.
 Data : off<EOF>
Sent 7 bytes to client.
```

Рисунок 5.5 – вивід консолі цільового агента з результатом виконання команди

На наведеній схемі (рисунок 5.6) подано алгоритм роботи реєстру мультиагентної системи, залучаючи два активних агента, та ввімкнений процес моніторингу системи на доступність агентів.

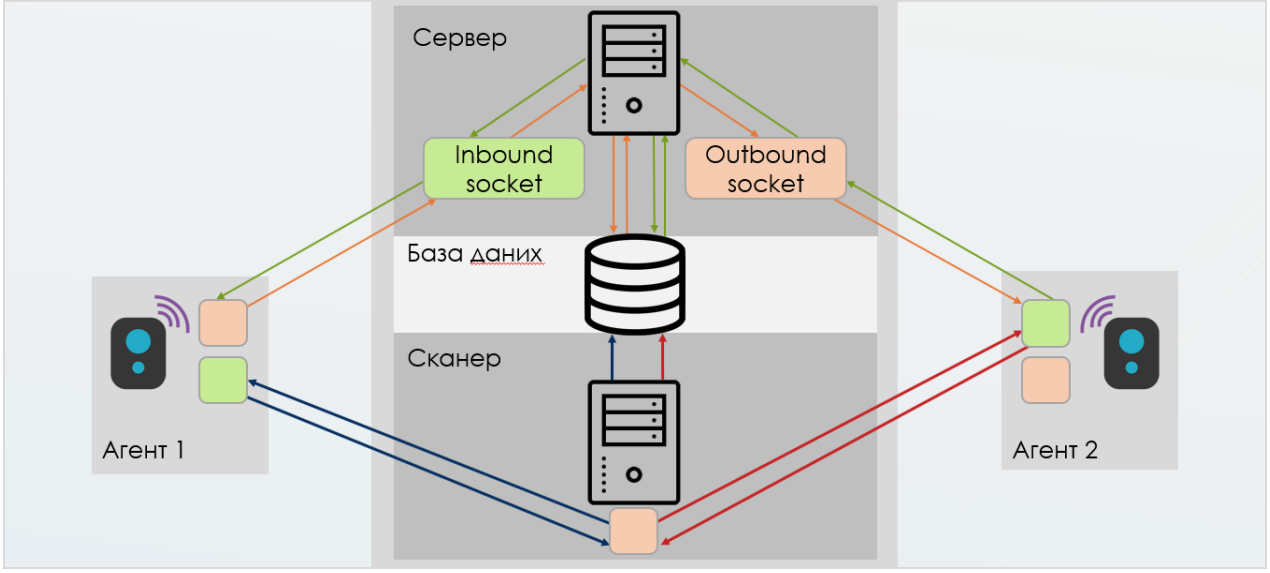

Рисунок 5.6 – схема роботи мультиагентної системи

Як видно із наведеної схеми – всі операції комунікації агентів записуються до журналу історії взаємодії між агентами мультиагентної системи.

ВИСНОВКИ

У ході виконання даної роботи було успішно розроблено програмний агент реєстру мультиагентної системи на базі платформи .NET Core. Система складається із сокет серверу за допомогою протоколу TCP. Перевагами даної системи можна вважати високий рівень стабільності, асинхронна обробка запитів від агентів, що дозволяє значно покращити продуктивність системи, а також ведення повної історії стану системи та всіх оброблених операцій.

Користувачами системи являються агенти мультиагентної системи. Інтерфейс користувача реалізовано за шаблоном консольного додатку, оскільки користувачі напряму із ним не взаємодіють а спілкування між агентами здійснюється за допомогою повідомлень на базі сокетів по каналу TCP у форматі символьних рядків за заздалегідь визначеним синтаксисом. Вхідною інформацією є рядок символів який транслюється у команду зрозумілу для сервера.

Враховуючи наведені вимоги до системи, програмний продукт розроблено з використанням технологій: Socket API, фреймворку Entity Framework Core,платформи .NET Core. Програма була розроблена на мові програмування C#, для кращої роботи системи з сервером MSSQL та кращої інтеграції додатку із сервером системи управління базами даних.

Серед основних функцій програмного забезпечення можна виділити:

– забезпечення стабільності функціонування реєстру MAC;
– зберігання історії виконаних операцій;
– ведення журналу помилок системи;
– забезпечення надійної комунікації агентів MAC за допомогою сервісу «Yellow Pages»;
– проведення моніторингу мережі на предмет доступності агентів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Рихтер Д. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж. Рихтер. – Санкт-Петербург: Питер, 2013. – 896 с. – (Мастер-класс).

2. Краткий обзор языка C# [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/.

3. TIOBE Index for C# [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.tiobe.com/tiobe-index/csharp/.

4. Шилдт Г. C#: полное руководство / Г. Шилдт. – Москва: Вильямс, 2011. – 1056с.

5. Албахари Д. C# 6.0 Справочник. Полное описание языка / Д. Албахари, Б. Албахари. – Москва: Вильямс, 2016. – 1040 с. – (6-е издание).

6. Программирование с использованием C# – Москва: Питер, 2006. – 432 с. – (Мастер-класс).

7. Microsoft Visual Studio 2015 [Electronic resource]. — Access mode: http://www.pcweek.com/infrastructure/article/detail.php?ID=173068

8. Pro C# 5.0 and the .NET 4.5 Framework, 6th edition/ Andrew Troelsen — Boston, The Facets of Ruby Series, 2013. – P.456

9. Common Type System [Electronic resource]. — Access mode: https://msdn.microsoft.com/en-us/library/zcx1eb1e(v=vs.110).aspx.

10. SQL Server 2014 Express edition [Electronic resource]. — Access mode: https://www.microsoft.com/en-us/server-cloud/products/sql-server.

11. The Repository Pattern [Electronic resource]. — Access mode: https://msdn.microsoft.com/en-us/library/ff649690.aspx.

12. Месарович М. Теория иерархических многоуровневых систем / Месарович М., Мако Д., Такахара И. – М. : Мир, 1973. – 344 с.

13. Каменнова М.С. Корпоративные информационные системы: технологии и решения / М.С. Каменнова // Системы управления базами данных. – 1995. – № 3. – С. 88-99.

14. Docs Microsoft // Asynchronous Client Socket Example [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/framework/network-programming/asynchronous-client-socket-example

15. Docs Microsoft // Asynchronous Server Socket Example [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/framework/network-programming/asynchronous-server-socket-example

16. Docs Microsoft // Sockets [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/framework/network-programming/sockets

17. Github // Technical documentation for Microsoft SQL Server [Електронний ресурс] — Режим доступу: https://github.com/MicrosoftDocs/sql-docs

18. Docs Microsoft // Design a DDD-oriented microservice [Електронний ресурс] — Режим доступу: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/ddd-oriented-microservice

19. Данілін А.В. Стандарти і єдина архітектура інформаційних технологій. — Microsoft Press, 2003. http://www.microsoft.com/Ukraine/Government/Analytics/IntegrationTechnologies/Standards.mspx.

20. SQL Server [Електронний ресурс] // Microsoft. – 2017. – Режим доступу до ресурсу: https://www.microsoft.com/ru-ru/sql-server/sql-server-2017-editions.

21. The MIT License (MIT) [Електронний ресурс] – Режим доступу до ресурсу: https://opensource.org/licenses/MIT.

22. GitHub vs. Bitbucket vs. GitLab vs. Coding [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://medium.com/flow-ci/github-vs-bitbucket-vs-gitlab-vs-coding-7cf2b43888a1.

23. User Datagram Protocol [Electronic resource]. — Access mode: https://technet.microsoft.com/en-us/library/cc785220.

24. DDF (Dynamic Data Flow — Динамичеcкий поток данных) [Элeктронный рeсурс]. — Рeжим доступа: http://www.locarus.ru/pages/products/tehnologii/ddf

ДОДАТОК 1

Програмний агент реєстру мультиагентної системи

Специфікація

УКР.НТУУ“КПІ”.ТМ51104_19Б

Аркушів 2

2019

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ«КПІ ім. Ігоря Сікорського».ТМ51104_19Б 81-1	Записка	Пояснювальна записка
Компоненти		
УКР.НТУУ«КПІ». ТМ51104_19Б 12-1	Текст програмного модулю	
УКР.НТУУ«КПІ». ТМ51104_19Б 13-1	Опис програми	

ДОДАТОК 2

Програмний агент реєстру мультиагентної системи

Текст програмного модулю

УКР.НТУУ“КПІ”.ТМ51104_19Б 12-1

Аркушів 10

2019

```
namespace AgentRegistry.Server.Listener
{
    public class AsynchronousSocketListener
    {
        public static ManualResetEvent allDone = new ManualResetEvent(false);
        public static int ListeningPort = 11000;
        public AsynchronousSocketListener()
        {
        }
        public static void StartListening()
        {
            IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
            IPAddress ipAddress = ipHostInfo.AddressList.First(x => x.AddressFamily ==
AddressFamily.InterNetwork);
            IPEndPoint localEndPoint = new IPEndPoint(ipAddress, ListeningPort);
            Socket listener = new Socket(ipAddress.AddressFamily,
                SocketType.Stream, ProtocolType.Tcp);
            try
            {
                listener.Bind(localEndPoint);
                listener.Listen(100);
                while (true)
                {
                    allDone.Reset();
                    Console.WriteLine($"Listening on port {ListeningPort}");
                    listener.BeginAccept(
                        callback: new AsyncCallback(AcceptCallback),
                        state: listener);
                    allDone.WaitOne();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
            Console.WriteLine("\nPress ENTER to continue...");
            Console.Read();
        }
        public static void AcceptCallback(IAsyncResult ar)
        {
            allDone.Set();
            var listener = (Socket)ar.AsyncState;
            var handler = listener.EndAccept(ar);
            var state = new StateObject
            {
                WorkSocket = handler
            };
            handler.BeginReceive(
                buffer: state.Buffer,
                offset: 0,
                size: StateObject.BufferSize,
                socketFlags: 0,
                callback: new AsyncCallback(ReadCallback),
                state: state);
        }
        public static void ReadCallback(IAsyncResult ar)
        {
            var content = string.Empty;
```

```
            var state = (StateObject)ar.AsyncState;
            var handler = state.WorkSocket;
            int bytesRead = handler.EndReceive(ar);
            if (bytesRead > 0)
            {
                state.StringBuilder.Append(Encoding.ASCII.GetString(state.Buffer, 0,
bytesRead));
                content = state.StringBuilder.ToString();
                if (content.IndexOf("<EOF>", StringComparison.Ordinal) > -1)
                {
                    Console.WriteLine("Read {0} bytes from socket. \n Data : {1}",
                        content.Length, content);
                    var command = content.Replace("<EOF>", string.Empty);
                    var response = string.Empty;
                    SystemHelper.TryCatchDefault(() =>
                    {
                        response = CommunicationManager.HandleCommand(command);
                    });
                    Send(handler, response + "<EOF>");
                }
                else
                {
                    handler.BeginReceive(state.Buffer, 0, StateObject.BufferSize, 0,
                    new AsyncCallback(ReadCallback), state);
                }
            }
        }
        private static void Send(Socket handler, String data)
        {
            byte[] byteData = Encoding.ASCII.GetBytes(data);
            handler.BeginSend(byteData, 0, byteData.Length, 0, new
AsyncCallback(SendCallback), handler);
        }
        private static void SendCallback(IAsyncResult ar)
        {
            try
            {
                Socket handler = (Socket)ar.AsyncState;
                int bytesSent = handler.EndSend(ar);
                Console.WriteLine("Sent {0} bytes to client.", bytesSent);
                handler.Shutdown(SocketShutdown.Both);
                handler.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
    }
}
namespace AgentRegistry.Server.Listener
{
    public class CommunicationManager
    {
        public class AvailableCommands
        {
            public const string Commands = "commands";
            public const string AgentsOnline = "agents_online";
```

```
            public const string AgentCommand = "agent_commands";
            public const string SendCommandTo = "send_command_to";
        }
        public static ReadOnlyCollection<string> AllCommands =>
            new ReadOnlyCollection<string>(new List<string>
            {
                AvailableCommands.Commands,
                AvailableCommands.AgentsOnline,
                AvailableCommands.AgentCommand,
                AvailableCommands.SendCommandTo
            });
        public static string HandleCommand(string command)
        {
            if (!AllCommands.Any(x => x == command.Split(" ").First()))
                return "Unknown_Command";
            if (command.StartsWith(AvailableCommands.Commands))
            {
                return HandleCommands();
            }
            else if (command.StartsWith(AvailableCommands.AgentsOnline))
            {
                return HandleAgentsOnline();
            }
            else if (command.StartsWith(AvailableCommands.AgentCommand))
            {
                return HandleAgentCommand(command);
            }
            else if (command.StartsWith(AvailableCommands.SendCommandTo))
            {
                return HandleSendCommandTo(command);
            }
            else
            {
                return "Unknown_Command";
            }
        }
        private static string HandleCommands()
            => $"Available
commands:{Environment.NewLine}{string.Join(Environment.NewLine, AllCommands)}";
        private static string HandleAgentsOnline()
        {
            var availableAgents = new List<AvailableAgentDTO>();
            availableAgents = new
SystemManager(Common.DataContext).GetAvailableAgents();
            if (!availableAgents.Any())
                return "No agents available";
            return availableAgents.Aggregate($"{Environment.NewLine}Available
agents:{Environment.NewLine}", (result, agent) => result + $"Agent Id -
{agent.AgentId}; Agent Type - {agent.AgentType}; IP Address - {agent.IpAddress}; Port -
{agent.Port}{Environment.NewLine}");
        }
        private static string HandleAgentCommand(string command)
        {
            var agentType = command.Split(" ").Last();
            var agentCommands = new
SystemManager(Common.DataContext).GetAgentCommands(agentType);
```

```
            return agentCommands.Aggregate($"{Environment.NewLine}Available {agentType}
agent commands:{Environment.NewLine}", (result, agentCommand) => result + $"Command -
{agentCommand};{Environment.NewLine}");
        }
        private static string HandleSendCommandTo(string command)
        {
            var commandParams = TranslateCommandToAgent(command);
            var idCommand = new
SystemManager(Common.DataContext).CreateCommandToAgent(commandParams);
            var responseCode = new
AsynchronousClient().SendMessage(commandParams.ToPort,
commandParams.CommandName).Replace("<EOF>", string.Empty);
            new SystemManager(Common.DataContext).CompleteAgentCommand(idCommand,
responseCode);
            return responseCode;
        }
        private static CommandToAgentDTO TranslateCommandToAgent(string command)
        {
            var commandParams = command.Split(" ");
            return new CommandToAgentDTO
            {
                Ip = commandParams[1],
                ToPort = Convert.ToInt32(commandParams[2]),
                CommandName = commandParams[3],
                FromPort = Convert.ToInt32(commandParams[4])
            };
        }
    }
}
namespace AgentRegistry.Scanner
{
    public class PortScanner
    {
        private const int MinPort = 1;
        private const int MaxPort = 65535;
        private readonly IEnumerable<int> _portsToScan = new List<int>();
        private List<OpenPortDTO> _openPorts = new List<OpenPortDTO>();
        private List<int> _closedPorts = new List<int>();
        public ReadOnlyCollection<OpenPortDTO> OpenPorts => new
ReadOnlyCollection<OpenPortDTO>(_openPorts);
        public ReadOnlyCollection<int> ClosedPorts => new
ReadOnlyCollection<int>(_closedPorts);
        public static string Host { get; } =
Dns.GetHostEntry(Dns.GetHostName()).AddressList.First(x => x.AddressFamily ==
AddressFamily.InterNetwork).ToString();
        public PortScanner(int minPort, int maxPort)
        {
            if (minPort > maxPort)
                throw new ArgumentException("Min port cannot be greater than max
port");
            if (minPort < MinPort || minPort > MaxPort)
                throw new ArgumentOutOfRangeException($"Min port cannot be less than
{MinPort} or greater than {MaxPort}");
            if (maxPort < MinPort || maxPort > MaxPort)
                throw new ArgumentOutOfRangeException($"Max port cannot be less than
{MinPort} or greater than {MaxPort}");
            _portsToScan = Enumerable.Range(minPort, maxPort - minPort);
            SetupLists();
```

```
        }
        public PortScanner(IEnumerable<int> portsToScan)
        {
            if (portsToScan.Any(x => x < MinPort) || portsToScan.Any(x => x > MaxPort))
                throw new ArgumentOutOfRangeException($"Port cannot be less than
{MinPort} or greater than {MaxPort}");
            _portsToScan = new List<int>(portsToScan);
            SetupLists();
        }
        public static async Task RunPortScanAsync(int minPort, int maxPort)
        {
            Console.WriteLine($"Checking ports {minPort} - {maxPort} on
localhost...\n");
            var portScanner = new PortScanner(minPort, maxPort);
            var progress = new Progress<PortScanResult>();
            progress.ProgressChanged += (sender, args) =>
            {
                Console.WriteLine($"Port {args.PortNum} is {(args.IsPortOpen ? "open" :
"closed")}");
            };
            await portScanner.ScanAsync(progress);
            portScanner.LastPortScanSummary();
        }
        public static async Task<ScanResultDTO> RunPortScanAsync(IEnumerable<int>
portsToScan)
        {
            Console.WriteLine($"Checking specified range of ports on {Host}...\n");
            var portScanner = new PortScanner(portsToScan);
            var progress = new Progress<PortScanResult>();
            progress.ProgressChanged += (sender, args) =>
            {
                Console.WriteLine($"Port {args.PortNum} is {(args.IsPortOpen ? "open" :
"closed")}");
            };
            await portScanner.ScanAsync(progress);
            portScanner.LastPortScanSummary();
            return new ScanResultDTO(portScanner._openPorts, portScanner._closedPorts,
Host);
        }
        public void LastPortScanSummary()
        {
            string openPorts = !_openPorts.Any() ? _openPorts.Count.ToString() :
string.Join(",", _openPorts.Select(x => x.Port).ToList());
            string closedPorts = !_closedPorts.Any() ? _openPorts.Count.ToString() :
string.Join(",", _closedPorts);
            Console.WriteLine();
            Console.WriteLine("-----------------");
            Console.WriteLine("Port Scan Results");
            Console.WriteLine("-----------------");
            Console.WriteLine();
            Console.WriteLine($"Open Ports......: {openPorts}");
            Console.WriteLine($"Closed Ports....: {closedPorts}");
            Console.WriteLine();
        }
        public async Task<string> IsPortOpenAsync(int port)
        {
            Socket socket = null;
            try
```

```
            {
                socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
                await Task.Run(() => socket.Connect(Host, port));
                var response = new AsynchronousClient().SendMessage(port, "check",
releaseSockects: true);
                return response;
            }
            catch (SocketException ex)
            {
                if (ex.SocketErrorCode == SocketError.ConnectionRefused)
                    return string.Empty;
                Debug.WriteLine(ex.ToString());
                Console.WriteLine(ex);
            }
            finally
            {
                if (socket?.Connected ?? false)
                    socket?.Disconnect(false);
                socket?.Close();
            }
            return string.Empty;
        }
        private async Task CheckPortAsync(int port, IProgress<PortScanResult> progress)
        {
            string response = await IsPortOpenAsync(port);
            if (!string.IsNullOrWhiteSpace(response))
            {
                _openPorts.Add(new OpenPortDTO {
                    Port = port,
                    AgentType = response.Replace("<EOF>", "")
                });
                progress?.Report(new PortScanResult { PortNum = port, IsPortOpen = true
});
            }
            else
            {
                _closedPorts.Add(port);
                progress?.Report(new PortScanResult() { PortNum = port, IsPortOpen =
false });
            }
        }
        private async Task ScanAsync(IProgress<PortScanResult> progress)
        {
            foreach (var port in _portsToScan)
                await CheckPortAsync(port, progress);
        }
        private void SetupLists()
        {
            _openPorts = new List<OpenPortDTO>();
            _closedPorts = new List<int>();
        }
    }
}
namespace AgentRegistry.Client
{
    public class AsynchronousClient
    {
```

```
        private int _port;
        private ManualResetEvent connectDone = new ManualResetEvent(false);
        private ManualResetEvent sendDone = new ManualResetEvent(false);
        private ManualResetEvent receiveDone = new ManualResetEvent(false);
        private string response = string.Empty;
        private bool _releaseSockets;
        public string SendMessage(int port, string message, bool releaseSockects =
false)
        {
            try
            {
                _port = port;
                response = string.Empty;
                _releaseSockets = releaseSockects;
                IPHostEntry ipHostInfo = Dns.GetHostEntry(Dns.GetHostName());
                IPAddress ipAddress = ipHostInfo.AddressList.First(x => x.AddressFamily
== AddressFamily.InterNetwork);
                IPEndPoint remoteEP = new IPEndPoint(ipAddress, _port);
                Socket client = new Socket(ipAddress.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);
                client.BeginConnect(remoteEP, new AsyncCallback(ConnectCallback),
client);
                connectDone.WaitOne();
                Send(client, message + "<EOF>");
                sendDone.WaitOne();
                Receive(client);
                receiveDone.WaitOne();
                Console.WriteLine("Response received : {0}", response);
                if (releaseSockects)
                {
                    client.Shutdown(SocketShutdown.Both);
                    client.Close();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
            return response;
        }
        private void ConnectCallback(IAsyncResult ar)
        {
            try
            {
                Socket client = (Socket)ar.AsyncState;
                client.EndConnect(ar);
                Console.WriteLine($"Socket connected to
{client.RemoteEndPoint.ToString()}");
                connectDone.Set();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
        private void Receive(Socket client)
        {
            try
```

```
            {
                StateObject state = new StateObject();
                state.WorkSocket = client;
                client.BeginReceive(state.Buffer, 0, StateObject.BufferSize, 0, new
AsyncCallback(ReceiveCallback), state);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
        private void ReceiveCallback(IAsyncResult ar)
        {
            try
            {
                StateObject state = (StateObject)ar.AsyncState;
                Socket client = state.WorkSocket;
                int bytesRead = client.EndReceive(ar);
                if (bytesRead > 0)
                {
                    state.StringBuilder.Append(Encoding.ASCII.GetString(state.Buffer,
0, bytesRead));
                    client.BeginReceive(state.Buffer, 0, StateObject.BufferSize, 0,
                        new AsyncCallback(ReceiveCallback), state);
                }
                else
                {
                    if (state.StringBuilder.Length > 1)
                    {
                        response = state.StringBuilder.ToString();
                    }
                    receiveDone.Set();
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
        }
        private void Send(Socket client, String data)
        {
            byte[] byteData = Encoding.ASCII.GetBytes(data);
            client.BeginSend(byteData, 0, byteData.Length, 0,
                new AsyncCallback(SendCallback), client);
        }
        private void SendCallback(IAsyncResult ar)
        {
            try
            {
                Socket client = (Socket)ar.AsyncState;
                int bytesSent = client.EndSend(ar);
                Console.WriteLine("Sent {0} bytes to server.", bytesSent);
                sendDone.Set();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.ToString());
            }
```

```
        }
    }
}
namespace AgentRegistry.ApplicationLogic.System
{
    public class SystemManager
    {
        private readonly IDataContext dataContext;
        private readonly IScannerLogRepository scannerLogRepository;
        public SystemManager(IDataContext dataContext)
        {
            this.dataContext = dataContext;
            this.scannerLogRepository = new ScannerLogRepository(this.dataContext);
        }
        public int StartScanSession()
        {
            var scanSession = new ScannerLog
            {
                DateTimeScanStart = DateTime.Now
            };
            scannerLogRepository.Add(scanSession);
            dataContext.SaveChanges();
            return scanSession.Id;
        }
        public void CompleteScanSession(int idScanSession, ScanResultDTO scanResult)
        {
            var scanSession = scannerLogRepository.FindById(idScanSession);
            if (scanResult.OpenPorts.Any())
            {
                foreach (var item in scanResult.OpenPorts)
                {
                    scanSession.Agents.Add(new Agent
                    {
                        AgentType = dataContext.Set<AgentType>().First(x =>
x.AgentTypeName == item.AgentType),
                        Port = item.Port,
                        IpAddress = scanResult.IpAddress,
                        ScannerLog = scanSession
                    });
                }
            }
            scanSession.DateTimeScanEnd = DateTime.Now;
            scanSession.IsSuccess = true;
            scannerLogRepository.Update(scanSession);
        }
        public List<AvailableAgentDTO> GetAvailableAgents()
        {
            var scanSession = dataContext.Set<ScannerLog>()
                .Where(x => x.IsSuccess ?? false)
                .OrderByDescending(x => x.DateTimeScanEnd)
                .First();
            return scanSession.Agents.Select(x => new AvailableAgentDTO
            {
                AgentId = x.Id.ToString(),
                AgentType = x.AgentType.AgentTypeName,
                IpAddress = x.IpAddress,
                Port = x.Port.ToString()
            }).OrderBy(x => x.AgentType).ToList();
```

```
        }
        public List<string> GetAgentCommands(string agentTypeName)
        {
            var agentType = dataContext.Set<AgentType>().Single(x => x.AgentTypeName ==
agentTypeName);
            return agentType.Commands.Select(x => x.CommandName).ToList();
        }
        public int CreateCommandToAgent(CommandToAgentDTO commandParams)
        {
            var scanSession = dataContext.Set<ScannerLog>()
                .Where(x => x.IsSuccess ?? false)
                .OrderByDescending(x => x.DateTimeScanEnd)
                .First();
            var agentFrom = scanSession.Agents.First(x => x.IpAddress ==
commandParams.Ip && x.Port == commandParams.FromPort);
            var agentTo = scanSession.Agents.First(x => x.IpAddress == commandParams.Ip
&& x.Port == commandParams.ToPort);
            var command = dataContext.Set<AgentCommand>().First(x => x.AgentType.Id ==
agentTo.AgentType.Id && x.CommandName == commandParams.CommandName);
            var agentCommunicationLog = new AgentsCommunicationLog
            {
                AgentFrom = agentFrom,
                AgentTo = agentTo,
                DateTimeCommunication = DateTime.Now,
                Command = command
            };
            dataContext.Add(agentCommunicationLog);
            dataContext.SaveChanges();
            return agentCommunicationLog.Id;
        }
        public void CompleteAgentCommand(int idCommand, string responseCode)
        {
            var communicationLog = dataContext.Set<AgentsCommunicationLog>().Single(x
=> x.Id == idCommand);
            var response = dataContext.Set<AgentCommandResponse>().Single(x =>
x.AgentCommand.Id == communicationLog.Command.Id && x.ResponseCode == responseCode);
            communicationLog.IsSuccess = true;
            communicationLog.CommandResponse = response;
            dataContext.SaveChanges();
        }
    }
}
namespace AgentRegistry
{
    public static class SystemHelper
    {
        public static void TryCatchDefault(Action action)
        {
            try
            {
                if (action != null)
                {
                    action.Invoke();
                }
            }
            catch (Exception ex)
            {
                Common.BuildServices();
```

```
                Common.Logger.LogException(ex);
                throw;
            }
        }
    }
}
namespace AgentRegistry
{
    public static class IDataContextExtensions
    {
        public static void TryBeginCommitTransaction(this IDataContext dataContext,
Action action)
        {
            using (var tran = dataContext.Database.BeginTransaction())
            {
                SystemHelper.TryCatchDefault(() =>
                {
                    try
                    {
                        if (action != null)
                        {
                            action.Invoke();
                            dataContext.SaveChanges();
                            tran.Commit();
                        }
                    }
                    catch (Exception)
                    {
                        dataContext.Database.CurrentTransaction.Rollback();
                        throw;
                    }
                });
            }
        }
    }
}
namespace AgentRegistry.Bootstrapper.Logger
{
    public class Logger : ILogger
    {
        public void LogException(Exception ex)
        {
            if (ex != null)
            {
                Common.ExceptionLogRepository.Add(new ExceptionLog
                {
                    DateTimeLogging = DateTime.Now,
                    ErrorMessage = ex.Message,
                    StackTrace = ex.StackTrace,
                    InnerExceptionMessage = ex.InnerException?.Message,
                    InnerExceptionStackTrace = ex.InnerException?.StackTrace
                });

                Common.DataContext.SaveChanges();
            }
        }
    }
}
```

ДОДАТОК З

Програмний агент реєстру мультиагентної системи

Опис програмного модулю

УКР.НТУУ“КПІ”.ТМ51104_19Б 13-1

Аркушів 6

2019

АНОТАЦІЯ

Розроблений програмний агент реєстру мультиагентної системи має функції забезпечення стабільності функціонування та відмовостійкості МАС, ведення журналів виконаних операцій та історії помилок, забезпечення комунікації агентів МАС, моніторингу мережі на доступність агентів та внесення їх до реєстру.

Потенційними сферами застосування даної системи можуть бути мультиагентні системи які в якості каналу зв’язку між агентами використовуються сокети та мережевий протокол TCP.

ЗМІСТ

1. ВІДОМОСТІ ПРО ПРОГРАМНИЙ МОДУЛЬ

При створенні програмного забезпечення були використані такі засоби реалізації:

– Середовище розробки Microsoft Visual Studio 2019;

– Мова програмування C#;

– Фреймворк .NET Core для організації архітектури додатку;

– Шаблон проектування архітектури додатку Domain Driven Design;

– Шаблон побудови додатків Console Application;

– Технологія Entity Framework Core для облегшеної організації доступу до бази даних;

– Програмний інтерфейс Socket API для організації комунікації між агентами;

– Мережевий протокол TCP ;

– Систему контролю версій Git;

– Систему управління реляційними базами даних Microsoft SQL Server, вона має безкоштовні версії для невеликих додатків та для етапів розробки програмного забезпечення та легко інтегрується з обраними технологіями.

1.1 Опис логічної структури

Для реалізації задачі створення програмного агента реєстру мультиагентної системи розроблено консольний серверний додаток із локальною базою даних, яка містить інформацію про стан системи, доступних агентів та журнали історії комунікації агентів та помилок системи.

Інтерфейс користувача виконано з використанням шаблону консольного додатку, мову програмування C# – для написання логіки системи.

1.2 Вхідні та вихідні дані

Вхідною інформацією є рядок символів який транслюється у команду зрозумілу для сервера.

Вихідною інформацією є результат виконаної команди на сервері чи іншому агенті. Якщо команда адресована іншому пристрою, сервер повертає код відповіді та, якщо передбачено, результат виконання команди. Якщо сервер отримав команду яка не підтримується він повертає повідомлення з відповідним текстом. У разі виникнення непередбачуваної помилки сервер повертає повідомлення з її кодом та описом, а також доповнює журнал помилок.

2. ВИКОРИСТАНІ ТЕХНІЧНІ ЗАСОБИ

Програмний модуль було протестовано на персональному комп’ютері, який працює на базі процесору x64 Intel Core i3 другого покоління та має 16 Гб оперативної пам’яті. Розроблене програмне забезпечення працює на комп’ютерах з операційною системою Windows, Linux, MacOS.